



Exasol User Manual

Version 6.0.10

**Empowering
analytics.**

Experience the world's fastest,
most intelligent, in-memory analytics
database.

Copyright © 2019 Exasol AG. All rights reserved.

The information in this publication is subject to change without notice. EXASOL SHALL NOT BE HELD LIABLE FOR TECHNICAL OR EDITORIAL ERRORS OR OMISSIONS CONTAINED HEREIN NOR FOR ACCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM THE FURNISHING, PERFORMANCE, OR USE OF. No part of this publication may be photocopied or reproduced in any form without prior written consent from Exasol. All named trademarks and registered trademarks are the property of their respective owners.

Table of Contents

Foreword	ix
Conventions	xi
Changes in Version 6.0	xiii
1. What is Exasol?	1
2. SQL reference	5
2.1. Basic language elements	5
2.1.1. Comments in SQL	5
2.1.2. SQL identifier	5
2.1.3. Regular expressions	7
2.2. SQL statements	11
2.2.1. Definition of the database (DDL)	12
2.2.2. Manipulation of the database (DML)	37
2.2.3. Access control using SQL (DCL)	60
2.2.4. Query language (DQL)	72
2.2.5. Verification of the data quality	82
2.2.6. Other statements	87
2.3. Data types	103
2.3.1. Overview of Exasol data types	104
2.3.2. Data type details	104
2.3.3. Data type aliases	107
2.3.4. Type conversion rules	108
2.3.5. Default values	110
2.3.6. Identity columns	112
2.4. Geospatial data	113
2.4.1. Geospatial objects	114
2.4.2. Geospatial functions	115
2.5. Literals	117
2.5.1. Numeric literals	118
2.5.2. Boolean literals	119
2.5.3. Date/Time literals	119
2.5.4. Interval literals	119
2.5.5. String literals	121
2.5.6. NULL literal	121
2.6. Format models	121
2.6.1. Date/Time format models	122
2.6.2. Numeric format models	124
2.7. Operators	126
2.7.1. Arithmetic Operators	127
2.7.2. Concatenation operator 	128
2.7.3. CONNECT BY Operators	129
2.8. Predicates	129
2.8.1. Introduction	130
2.8.2. List of predicates	130
2.9. Built-in functions	135
2.9.1. Scalar functions	136
2.9.2. Aggregate functions	140
2.9.3. Analytical functions	140
2.9.4. Alphabetical list of all functions	143
3. Concepts	257
3.1. Transaction management	257
3.1.1. Basic concept	257
3.1.2. Differences to other systems	258
3.1.3. Recommendations for the user	258
3.2. Rights management	258
3.2.1. User	259

3.2.2. Roles	259
3.2.3. Privileges	260
3.2.4. Access control with SQL statements	260
3.2.5. Meta information on rights management	261
3.2.6. Rights management and transactions	261
3.2.7. Example of rights management	261
3.3. Priorities	262
3.3.1. Introduction	263
3.3.2. Priorities in Exasol	263
3.3.3. Example	264
3.4. ETL Processes	264
3.4.1. Introduction	265
3.4.2. SQL commands IMPORT and EXPORT	265
3.4.3. Scripting complex ETL jobs	266
3.4.4. User-defined IMPORT using UDFs	266
3.4.5. User-defined EXPORT using UDFs	268
3.4.6. Hadoop and other systems	268
3.4.7. Using virtual schemas for ETL	269
3.4.8. Definition of file formats (CSV/FBV)	269
3.5. Scripting	271
3.5.1. Introduction	272
3.5.2. General script language	273
3.5.3. Database interaction	280
3.5.4. Libraries	287
3.5.5. System tables	294
3.6. UDF scripts	294
3.6.1. What are UDF scripts?	295
3.6.2. Introducing examples	296
3.6.3. Details for different languages	301
3.6.4. The synchronous cluster file system BucketFS	323
3.6.5. Expanding script languages using BucketFS	325
3.7. Virtual schemas	329
3.7.1. Virtual schemas and tables	330
3.7.2. Adapters and properties	331
3.7.3. Grant access on virtual tables	332
3.7.4. Privileges for administration	332
3.7.5. Metadata	333
3.7.6. Details for experts	334
3.8. SQL Preprocessor	335
3.8.1. How does the SQL Preprocessor work?	336
3.8.2. Library sqlparsing	336
3.8.3. Best Practice	339
3.8.4. Examples	339
3.9. Profiling	345
3.9.1. What is Profiling?	346
3.9.2. Activation and Analyzing	346
3.9.3. Example	347
3.10. Skyline	348
3.10.1. Motivation	349
3.10.2. How Skyline works	349
3.10.3. Example	350
3.10.4. Syntax elements	350
4. Clients and interfaces	353
4.1. EXAplus	353
4.1.1. Installation	353
4.1.2. The graphical user interface	354
4.1.3. The Console mode	358
4.1.4. EXAplus-specific commands	361

4.2. ODBC driver	377
4.2.1. Supported standards	378
4.2.2. Windows version of the ODBC driver	378
4.2.3. Linux/Unix version of the ODBC driver	381
4.2.4. Establishing a connection in an ODBC application	382
4.2.5. Connecting via Connection Strings	383
4.2.6. Support for character sets	385
4.2.7. Best Practice for developers	386
4.3. JDBC driver	386
4.3.1. Supported standards	387
4.3.2. System requirements	387
4.3.3. Using the driver	388
4.3.4. Best Practice for developers	392
4.4. ADO.NET Data Provider	392
4.4.1. Supported standards, system requirements and installation	393
4.4.2. Using the Data Provider	393
4.4.3. Exasol Data Destination	397
4.4.4. Exasol Data Processing Extension	399
4.5. WebSockets	399
4.6. SDK	400
4.6.1. The call level interface (CLI)	400
A. System tables	405
A.1. General information	405
A.2. List of system tables	405
A.2.1. Catalog system tables	405
A.2.2. Metadata system tables	405
A.2.3. Statistical system tables	438
A.2.4. System tables compatible with Oracle	456
B. Details on rights management	459
B.1. List of system and object privileges	459
B.2. Required privileges for SQL statements	461
B.3. System tables for rights management	465
C. Compliance to the SQL standard	467
C.1. SQL 2008 Standard Mandatory Features	467
C.2. SQL 2008 Standard Optional Features	472
D. Supported Encodings for ETL processes and EXAplus	473
E. Customer Service	475
Abbreviations	477
Index	481

List of Tables

2.1. Pattern elements in regular expressions	9
2.2. Overview of Exasol data types	104
2.3. Summary of Exasol aliases	108
2.4. Possible implicit conversions	109
2.5. Elements of Date/Time format models	123
2.6. Elements of numeric format models	125
2.7. Precedence of predicates	130
3.1. Additional basics for UDF scripts	296
4.1. Information about the work with EXAplus	356
4.2. EXAplus command line parameters (Console mode only)	359
4.3. Known problems associated with using the ODBC driver on Windows	381
4.4. Known problems when using the ODBC driver for Linux/Unix	382
4.5. Supported DriverProperties of the JDBC driver	391
4.6. Keywords in the ADO.NET Data Provider connection string	394
B.1. System privileges in Exasol	460
B.2. Object privileges in Exasol	461
B.3. Required privileges for running SQL statements	462
C.1. SQL 2008 Mandatory Features	468
C.2. SQL 2008 Optional Features supported by Exasol	472

Foreword

This User Manual provides an overview of Exasol and documents the user interfaces and the extent of which the [SQL](#) language is supported. Further technical information about Exasol can also be found in our [Online Solution Center](#) [<https://wwwexasolcom/portal/display/SOL>].

We always strive to ensure the highest possible quality standards. With this in mind, Exasol warmly invites you to participate in improving the quality of documentation.

Therefore, please inform us if you:

- are missing information
- have found errors in the documentation
- think sections are unclear or too brief
- think the examples are not conclusive enough, or insufficient.

Send us your suggestions and comments to the address shown below or add them directly online in our [IDEA project](#) [<https://wwwexasolcom/support/projects/IDEA/issues/>]. We thank you sincerely and will endeavor to implement your suggestions in the next version.



Further details about our support can be found in our [Support Dashboard](#) [<https://wwwexasolcom/portal/display/EXA/Support+Dashboard>]

Conventions

Symbols

In this User Manual, the following symbols are used:



Note: e.g. "Please consider that inside the database, empty strings are interpreted as NULL values."



Tip: e.g. "We recommend to use local variables to explicitly show the variable declaration."



Important: e.g. "Regular identifiers are not case sensitive."



Caution: e.g. "The execution time of this command can take some time."

Changes in Version 6.0

New features and improvements

- "EXASOL goes Open Source":
 - EXASOL publishes a whole range of technologies as open source projects, which specifically focus on the areas of integration, usage and analytic extensions for our commercial database. We started with this concept earlier by allowing you to upload any open source package for the UDF script languages. Our vision is to provide flexible frameworks for expanding EXASOL's functionality. The advantage for this approach is a more agile development and provisioning process in addition to the individually adjusted features that can be outfitted for your needs.
 - You'll find our open source GitHub repository at <https://www.github.com/EXASOL>. We'd be very happy if you would contribute to our open source community by using, extending and adding to our open-sourced tools. Of course we are open for similar new projects. Let's shape EXASOL's future jointly together.
 - In version 6.0, you will find many product features which contain open source elements that we will expand on later:
 - Adapters to external data sources for implementing the concept of [virtual schemas](#) [<https://github.com/EXASOL/virtual-schemas>]
 - [UDF scripts](#) [<https://github.com/EXASOL/hadoop-etl-udfs>] to load data easily from Hadoop systems using the IMPORT FROM SCRIPT command (see also section ETL below)
 - Native drivers for our new, open client/server protocol "[JSON over WebSockets](#)" [<https://github.com/EXASOL/websocket-api>]
 - API specification for integrating new [script languages](#) [<https://github.com/EXASOL/script-languages>] and ready-to-use script containers (e.g. C++)
 - An exemplary usage of the [XMLRPC interface](#) [<https://github.com/EXASOL/exaoperation-xmlrpc>] to install and operate EXASOL automatically without the usage of the web interface EXAoperation
 - Projects to integrate EXASOL into abstraction frameworks, such as e.g. SQLAlchemy, Django or Teiid
 - SQL scripts to [migrate data](#) [<https://github.com/EXASOL/database-migration>] from other databases (e.g. Oracle, DB2, MS SQLServer, Teradata, MySQL, Postgres, Redshift, Vertica) to EXASOL by using lua scripting for automatically creating and executing the appropriate IMPORT statements
 - EXASOL created a public Maven repository (<https://maven.exasol.com>) which includes our JDBC driver and the script API definition. By including that repository, you can easily develop Java code for EXASOL scripts directly in your Java IDE.
 - A [Nagios monitoring Docker container](#) [<https://github.com/EXASOL/nagios-monitoring>] which can be installed within a minute. This is a simple starting point for setting up a running monitoring system for your EXASOL database, or for extracting the resulting Nagios configuration files into your existing monitoring tool.
 - Further documentation and usage tips can be found in the corresponding open source project, but also in our Online Solution Center: <https://wwwexasolcom/portal/display/SOL>
 - Product editions and feature sets
 - With version 6, EXASOL is introducing two editions, **Standard Edition**, the **Advanced Edition**. The Advanced Edition consists of the additional features customers previously ordered separately such as:
 - Virtual schemas (see next point)
 - UDF scripts (see [Section 3.6, “UDF scripts”](#))
 - Geospatial data (see [Section 2.4, “Geospatial data”](#))
 - Skyline (see [Section 3.10, “Skyline”](#))

You can now enjoy the following features without extra costs in the Standard Edition: Query Cache, LDAP authentication and IMPORT/EXPORT interfaces JDBC (JDBC data sources) and ORA (native Oracle interface).

If you have questions regarding these editions, please contact your EXASOL account manager or ask our support team.

- The new virtual schemas concept provides a powerful abstraction layer to conveniently access arbitrary data sources. Virtual schemas are a read-only link to an external source and contain virtual tables which

- look like regular tables except that the data is not stored locally. Details can be found in [Section 3.7, “Virtual schemas”](#).
- Using the new synchronous file system BucketFS, you can store files on the cluster and provide local access for UDF scripts. Details can be found in [Section 3.6.4, “The synchronous cluster file system BucketFS”](#).
 - EXASOL’s script framework was enhanced so that you can now install new script languages on the EXASOL cluster. You can either use several versions of a language (e.g. Python 2 and Python 3), add additional libraries (even for R), or integrated completely new languages (Julia, C++, ...). Further details can be found in [Section 3.6.5, “Expanding script languages using BucketFS”](#).
 - The option `IF NOT EXISTS` was introduced in statements `CREATE SCHEMA`, `CREATE TABLE` and `ALTER TABLE(column)`. You can execute DDL scripts automatically without receiving error messages when certain objects already exist.
 - `PRELOAD` loads certain tables or columns and the corresponding internal indices from disk in case they are not yet in the database cache. Due to EXASOL’s smart cache management we highly recommend to use this command only exceptionally. Otherwise you risk a worse overall system performance.
 - Similar to the structures in the UDF scripts, we added the metadata information to the scripting language. Additionally, this metadata was extended by the script schema and the current user. Please refer to the corresponding sections of [Section 3.5, “Scripting”](#) and [Section 3.6, “UDF scripts”](#) for further details.
 - UDF scripts can handle dynamic input and output parameters to become more flexible. For details see section [Dynamic input and output parameters](#) in [Section 3.6, “UDF scripts”](#).
 - In SCALAR UDF scripts, you can now combine an `EMITS` output in expressions within the select list without any need to nest it in a separate subselect anymore.
 - Function `APPROXIMATE_COUNT_DISTINCT` calculates the approximate number of distinct elements, at a much faster rate than `COUNT` function.
 - By using the new parameter `NLS_DATE_LANGUAGE` in function `TO_CHAR(datetime)` you can overwrite the session settings.
 - Function `NVL2` was added which is similar to `NVL`, but has an additional parameter for replacing the values which are not `NULL`.
 - The bitwise functions `BIT_LSHIFT`, `BIT_RSHIFT`, `BIT_LROTATE` and `BIT_RROTATE` were added. Additionally, the bit range was extended from 59 to 64 for the other bitwise functions.
 - The new session/system parameter `TIMESTAMP_ARITHMETIC_BEHAVIOR` defines the behavior for `+/-` operators:
 - `TIMESTAMP_ARITHMETIC_BEHAVIOR = 'INTERVAL'` - The difference of two datetime values is an interval, and when adding a decimal value to a timestamp, the number is rounded to an integer, so actually a certain number of full days is added
 - `TIMESTAMP_ARITHMETIC_BEHAVIOR = 'DOUBLE'` - The difference of two datetime values is a double, and when adding a decimal value to a timestamp, the fraction of days is added (hours, minutes, ...)
 - When adjusting priorities using the `GRANT PRIORITY` statement, the changes are applied immediately. Before, only newly created transactions were affected.
 - The default limit for the possible number of schema objects was increased from 50000 to 250000.
 - ETL processes:
 - With UDF scripts, you can easily load data from nearly any external data source. The new syntax `IMPORT FROM SCRIPT` makes it very convenient to load data from various systems with a simple command, once the appropriate scripts have been developed. Further details can be found in [Section 3.4.4, “User-defined IMPORT using UDFs”](#), and we provide open source scripts (see <https://www.github.com/EXASOL>) for integrating Hadoop systems into EXASOL easily. In the near future, we will add additional open source code for further products.
 - The statements `IMPORT` and `EXPORT` using JDBC connections now support Kerberos authentication. You can specify the Kerberos configuration and keytab data through the `IDENTIFIED BY` clause of the statements or the used connections (see also `CREATE CONNECTION`).
 - Please note that from now on, you have to manually install the Oracle Instant Client via EXAoperation if you want to use the native Oracle interface for `IMPORT/EXPORT` statements. The pre-installed JDBC drivers have been updated.
 - Performance improvements:
 - The connection establishment to the database has been significantly accelerated.
 - Metadata requests from drivers are faster now.

- Adding small amounts of data through [INSERT](#) (e.g. only a single row) is significantly faster due to an automatic, hybrid storage mechanism. The "latest" data is stored row-wise and automatically merged into the column-wise storage in the background. The user does not notice anything except a higher insert rate because less data blocks have to be committed to the disks.
- In scenarios with many parallel write transactions, the overall system throughput was improved.
- The optimization of [SELECT](#) statements which access the same view (or [WITH](#) clause) several times was improved. As results, in more cases than before the better alternative is chosen, whether such a view (or [WITH](#) clause) should be materialized upfront or not. Overall, the performance should improve although the optimizer does not always make the right decision due to the complexity of the subject. Customers who have explicitly deactivated that view optimization via the extra database parameter `-disableviewoptimization`, might want to try out if this is not necessary anymore.
- By optimizing the internal data transfer for distributed table rows, several operations will gain performance (such as [MERGE](#), [IMPORT](#) and cluster enlargements).
- The [MERGE](#) statement will accelerate for certain situations, e.g. if the target table is large or if source and target tables are not distributed by the [ON](#) condition columns ([DISTRIBUTE BY](#)).
- When deleting data, the affected rows are internally just marked as deleted, but not physically dropped yet. After reaching a certain threshold, this data is deleted and the table reorganized. This reorganization is faster due to advanced internal parallelization.
- Due to an intelligent replication mechanism small tables are no longer persistently replicated, but are kept incrementally in the cache. Consequently, [INSERT](#) operations on small tables are significantly faster.
- [GROUP BY](#) calculations with a very small number of groups can be faster in situations, because the method how the group aggregations are distributed across the cluster nodes has been improved.
- The execution of [BETWEEN](#) filters on DATE columns takes already advantage of existing internal indices to accelerate the computation. We extended this capability by supporting the data types [TIMESTAMP](#), [DECIMAL](#), [DOUBLE](#) and [INTERVAL](#).
- A smart recompress logic has been implemented so that only tables and columns are recompressed if a substantial improvement can be achieved. This is evaluated automatically on the fly, and will accelerate the [DELETE](#) statement since this internally invokes a recompression after a certain threshold of deleted rows. You can still enforce a full compression of a table using the new [ENFORCE](#) option of the [RECOMPRESS](#) command. Further, you can now specify certain columns in the [RECOMPRESS](#) statement to selectively recompress only these columns.
- Instead of checking all constraints of a table, only the constraints for the new columns should be checked for statement [ALTER TABLE ADD COLUMN](#).
- For newly (!) created EXAStorage volumes, the read performance has been improved based on an optimized physical storage layout. Read operations on big and heavily fragmented data volumes are faster. This leads to an overall better read performance for situations where the main memory is not sufficient, leading to continuous data read from all kinds of different tables.
- Local backup and restore operations have been significantly accelerated. In internal tests we achieved a speedup factor between 4-5 for backup and 2-3 for restore operations. Please note that the actual speedup on your system depends on the specific hardware configuration.
- Smart recovery mechanisms reduce the amount of data which has to be restored by EXAStorage in case of node failures.
- The startup and restart times for databases has been accelerated, especially for larger clusters.
- System info:
 - You'll find lots of information about the usage of the various storage volumes in the system table [EXA_VOLUME_USAGE](#).
 - The [DURATION](#) column has been added in the [EXA_DB_Audit_SQL](#) and [EXA_SQL_LAST_DAY](#) system tables. It stores the duration of the statement in seconds.
 - The number of rows of a table can be found in column [TABLE_ROW_COUNT](#) of system tables [EXA_ALL_TABLES](#), [EXA_DB_A_TABLES](#) and [EXA_USER_TABLES](#).
- Interfaces:
 - Please note that with version 6.0 we dropped the support for AIX, HP-UX and Solaris. More details about our life cycle policies can be found here: <https://wwwexasolcom/portal/display/DOWNLOAD/EXASolution+Life+Cycle>
 - A connection-oriented JSON over WebSockets API has been introduced. It is now possible to integrate EXASOL with nearly any programming language of your choice. Another advantage is the protocol supports client/server compression. Initially, we have published an open-source EXASOL Python driver that uses

- this interface, but additional implementations for further languages will likely follow. Further details can be found in [Section 4.5, “WebSockets”](#).
- In case of severe overload situations, the user SYS can now analyze the system and terminate problematic processes via the new connection parameter SUPERCONNECTION (ODBC) or superconnection (JDBC and ADO.NET).
 - The parameter CONNECTTIMEOUT (ODBC) or connecttimeout (JDBC and ADO.NET) define the maximal time in milliseconds (default: 2000) the driver will wait to establish a TPC connection to a server. This timeout is intended to limit the overall login time especially in cases of a large cluster with several reserve nodes.
 - Encrypted client/server connections are now activated by default and based on the algorithm ChaCha20 (RFC 7539).
 - The evaluation of data types of prepared parameters in the PREPARE step has been improved. Until now, the generic type VARCHAR(2000000) was always used.
 - ODBC driver
 - If no schema is specified in the connection parameters, the driver will no longer try to open the schema with the corresponding user name. This behavior was implemented due to historical reasons, but was decided to be eliminated for consistency reasons.
 - JDBC
 - The JDBC driver added support for `java.sql.Driver` file. This option was added in the JDBC 4.0 standard which allows Java applications to use JDBC drivers without explicitly load a driver via `Class.forName` or `DriverManager.registerDriver`. It is now sufficient to add the driver to the classpath.
 - The data types of query result sets are optimized. In detail, function `getColumnType()` of class `ResultSetMetadata` determines the minimal matching integer type (`smallint`, `int` or `bigint`) for decimals without scale.
 - The JDBC driver supports parallel read and insert from the cluster nodes via so-called sub-connections. Details and examples can be found in our Solution Center: <https://wwwexasol.com/support/browse/SOL-546>
 - The class `EXAPreparedStatement` has the new method `setFixedChar()` which sets the parameter type to `CHAR`. Since the standard method `setString()` uses `VARCHAR`, this new function can be useful for comparisons with `CHAR` values.
 - ADO.NET driver
 - You can use the parameter `querytimeout` to define how many seconds a statement may run before it is automatically aborted.
 - The DDEX provider and pluggable SQL cartridge for SQL Server Analysis Services supports SQL Server 2016 (v 13.0).
 - EXAplus
 - The new graphical system usage statistics can be displayed by clicking on the pie chart button in the tool bar.
 - The bar at the bottom-right corner shows the memory allocation pool (heap) usage of the current EXAplus process. By double-clicking you can trigger the program to execute a garbage collection.
 - You can specify additional JDBC parameter using the commandline parameter `-jdbcparam` or the advanced settings in a connection profile.
 - The console version has new parameters to handle user profiles (`-lp` for printing existing profiles, `-dp <profile>` for deleting and `-wp <profile>` for writing certain connection options into a profile)
 - SDK
 - Until now we shipped two versions of our SDK library: `libexacl` and `libexacl_c`. The first library contained dependencies to C++, and due to a customer request we added the latter one without any dependencies in version 5. We removed the C++ dependencies from `libexacl` and will only ship this single version from now on.
 - Operation:
 - Details about how to upgrade to version 6 can be found in the following article in our Solution Center: <https://wwwexasol.com/support/browse/SOL-504>



After the upgrade to version 6, a downgrade to version 5 is only possible by a re-installation and a restore of an appropriate backup.



Due to performance reasons, we recommend to recreate the database volumes in EXAStorage, even though this means a longer downtime. If the volumes are not recreated, at every database a warning is raised that the old EXAStorage format is still used.

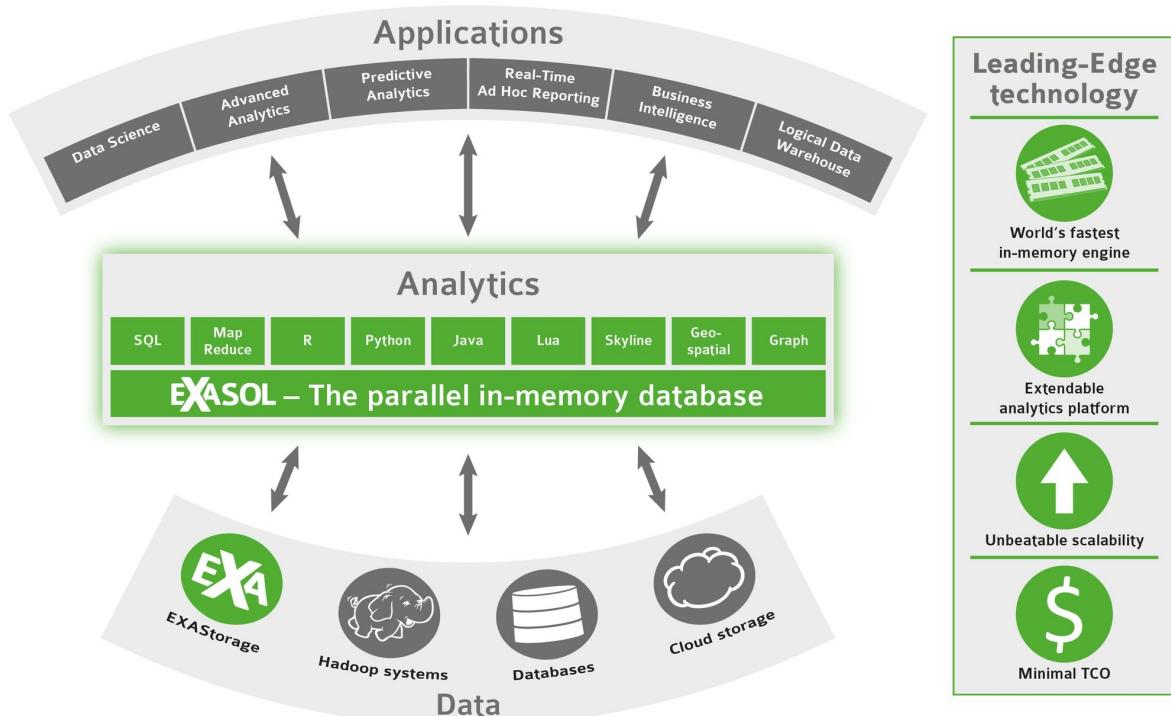
- In EXAStorage, it is possible to add disks without reinstalling the node.
- EXAStorage volumes can now have the state DEGRADED which indicates failed disks. Until now, the state was ONLINE in such situations as long as the data was still accessible (e.g. due to a RAID configuration).
- Persistent data volumes can be explicitly shrunk. This non-blocking shrink operation can be triggered via EXAoperation and you can either specify a certain target size or to shrink the volume as much as possible.
- We extended the XMLRPC administration interface so that the whole range of capabilities of the EXAoperation web interface is now available.
- It's possible to use WebHDFS as remote archive volumes for backups. Kerberos authentication is possible by specifying a delegation token in the options of the remote volume.
- Backups on remote archive volumes are automatically deleted if the expiration time has been exceeded. This type of cleanup is triggered after every backup run. Previously, this mechanism was only possible for local backups.
- Until now, the protocols UDP and TCP were supported for the communication with syslog servers. Now, we added support for TLS based, encrypted communication as described in RFC 5425.
- We added the priority of a message into the tag field for syslog servers which are defined in EXAoperation. Messages can now be classified easier since syslog prints this tag as a prefix to the actual message. The new prefix will therefore look like "EXAoperation [Priority], [Service]" instead of "EXAoperation, [Service]".
- The IP range for the internal private cluster network can now be configured (with limitations). Details can be found in the roadmap issue [EXASOL-1750](#) [<https://wwwexasol.com/support/browse/EXASOL-1750>].
- The license server can be installed via DVD image or semi-automated over via network. Now with version 6.0, it is also possible to do the latter fully automated. Additionally, EXASOL can be installed from a USB device (see also [EXASOL-1792](#) [<https://wwwexasol.com/support/browse/EXASOL-1792>])
- The access to the Support section under EXAoperation where you can download the debug information can now be configured via access privileges. Before, only users with the Administrator or Master role were able to do so. Please be aware that this debug log files can contain context-dependent sensitive information such as SQL statements.
- A new configuration field was added in EXAoperation monitoring section to specify the default log retention period for database log files.
- The access to the Support area in EXAoperation can now be granted through roles. Before, you had to be an administrator.
- In EXAoperation, it's possible to upload own TLS certificates.

Chapter 1. What is Exasol?



Exasol is the technology of choice for all kinds of analytic challenges in today's business world. From business-critical operational data applications and interactive advanced customer analytics systems to enterprise (logical) data warehouses - Exasol helps you attain optimal insights from all sorts of big data using a completely new level of analytical capabilities.

With Exasol you are enabled to extend your historical data frame, increase the precision of your analysis, satisfy the user experience through real-time responsiveness and multiply the number of users.



The deep integration of parallel R, Python, Java and Lua (see [Section 3.6, “UDF scripts”](#)) and the ability to combine regular SQL queries with Map Reduce jobs directly within our in-memory database offers a very powerful and

flexible capability for optimal in-database analytics. Graph analytics capabilities (see CONNECT BY in [SELECT](#)), our Skyline extension (see [Section 3.10, “Skyline”](#)) and the support of geospatial analysis (see [Section 2.4, “Geospatial data”](#)) completes the analytical spectrum.

Exasol combines in-memory, column-based and massively parallel technologies to provide unseen performance, flexibility and scalability. It can be smoothly integrated into every IT infrastructure and heterogeneous analytical platform.

Please be aware that Exasol is available in two different editions. Compared to the **Standard Edition**, the **Advanced Edition** contains the following extra features:

- Virtual schemas (see [Section 3.7, “Virtual schemas”](#))
- UDF scripts (see [Section 3.6, “UDF scripts”](#))
- Geospatial data (see [Section 2.4, “Geospatial data”](#))
- Skyline (see [Section 3.10, “Skyline”](#))

Interfaces

Currently, there are three important standardized interfaces for which Exasol makes appropriate driver software available: [ODBC](#), [JDBC](#) and [ADO.NET](#). In addition, the OLE DB Provider for ODBC Drivers™ from Microsoft allows Exasol to be accessed via an [OLE DB](#) interface.

Our [JSON](#) over WebSockets API allows you to integrate Exasol into nearly any programming language of your choice. Drivers built on top of that protocol, such as a native Python package, are provided as open source projects. And in order to be able to connect Exasol to in-house C/C++ applications, Exasol additionally provides a client [SDK](#) (Software Development Kit).

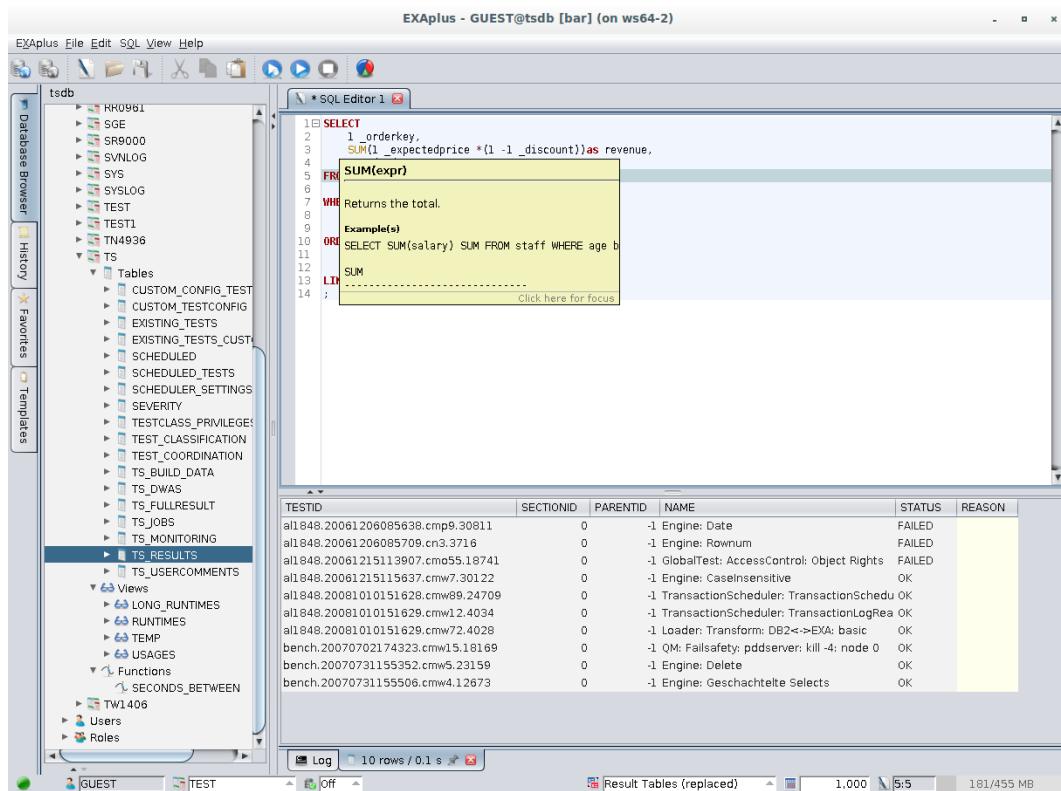
The driver connection makes it possible to send SQL queries and display the results in the connected application as well as to load data into Exasol. However, for reasons of performance the [IMPORT](#) statement is recommended for larger load operations.

Interface	Chapter in manual
ODBC driver	Section 4.2, “ODBC driver”
JDBC driver	Section 4.3, “JDBC driver”
ADO.NET Data Provider	Section 4.4, “ADO.NET Data Provider”
JSON over WebSockets	Section 4.5, “WebSockets”
Client SDK	Section 4.6, “SDK”

EXAplus

EXAplus is a console for entering [SQL](#) statements. EXAplus simplifies the task of running SQL scripts or performing administrative tasks. EXAplus was implemented in Java; hence, it is not platform-dependent and can be used as both a graphical user interface or text console.

Detailed documentation can be found in [Section 4.1, “EXAplus”](#).



Chapter 2. SQL reference

This [SQL](#) reference documents the majority of the extent of the SQL language supported by Exasol. Certain features are still to be documented, these will be included in future versions.

A detailed summary of the supported SQL standard features can be found in [Appendix C, Compliance to the SQL standard](#).

2.1. Basic language elements

2.1.1. Comments in SQL

Comments are primarily used to facilitate the clarity and maintainability of SQL scripts. They can be added at the discretion of the user anywhere in the SQL script.

There are two different kinds of comment in Exasol:

- Line comments begin with the character -- and indicate that the remaining part of the current line is a comment.
- Block comments are indicated by the characters /* and */ and can be spread across several lines. All of the characters between the delimiters are ignored.

Examples

```
-- This is a comment
SELECT * FROM dual;

-- Display the number of rows of table dual
SELECT count(*) FROM dual;

SELECT avg(salary)      -- Average salary
FROM staff              -- From table staff
WHERE age>50;          -- Limitation to over 50 years old

/* Any number of comment rows can be
   inserted between the delimiters
*/
SELECT * FROM dual;

SELECT /*This section is not evaluated!*/ * FROM dual;
```

2.1.2. SQL identifier

Each database object has a unique name. Within a SQL statement, these names are referenced via SQL identifiers. Database objects include schemas, tables, views, columns, functions, users, and roles. In Exasol SQL identifiers can have a maximum length of 128 characters. One differentiates between regular identifiers and identifiers in quotation marks.

Regular identifiers

Regular identifiers are stated without quotation marks. As defined in the SQL standard, Exasol only allows a subset of the unicode characters, whereas the first letter is more restricted than the following ones. The detailed list of characters of the specified classes are defined in the unicode standard.

- **First letter:** all letters of unicode classes Lu (upper-case letters), Ll (lower-case letters), Lt (title-case letters), Lm (modifier letters), Lo (other letters) and Nl (letter numbers).
- **Following letters:** the classes of the first letter (see above) and additionally all letters of unicode classes Mn (nonspacing marks), Mc (spacing combining marks), Nd (decimal numbers), Pc (connector punctuations), Cf (formatting codes) and the unicode character U+00B7 (middle dot).

Regarding e.g. the simple ASCII character set, these rules denote that a regular identifier may start with letters of the set {a-z,A-Z} and may further contain letters of the set {a-z,A-Z,0-9,_}.

A further restriction is that reserved words (see also below) cannot be used as a regular identifier.



If you want to use characters which are prohibited for regular identifiers, you can use delimited identifiers (see next section). E.g. if you want to use the word `table` as identifier, you have to quote it ("`table`"), since it's a reserved word.

Regular identifiers are always stored in upper case. They are not *case sensitive*. By way of example, the two identifiers are therefore equal ABC and aBc.



Regular identifiers are stored in upper case.

Examples

Identifier	Name in the DB
ABC	ABC
aBc	ABC
a123	A123

Delimited identifiers

These identifiers are names enclosed in double quotation marks. Any character can be contained within the quotation marks except the dot (' . '). Only in case of users and roles, this point is allowed for being able to use e.g. email addresses. This is possible because users and roles are not in conflict with schema-qualified objects (see next chapter).

A peculiarity arises from the use of double quotation marks: if a user wishes to use this character in the name, he must write two double quotation marks next to one another (i.e. "`ab" "c`" indicates the name `ab" c`).

Excepting users and roles, identifiers in quotation marks are always *case sensitive* when stored in the database.



Delimited identifiers are case-sensitive (except for users and roles).

Examples

Identifier	Name in the DB
"ABC"	ABC
"abc"	abc
"_x_"	_x_
"ab" "c"	ab" c

Schema-qualified names

If database objects are called via schema-qualified names, the names are separated with a point. This enables access to schema objects, which are not located in the current schema.

Examples

```
SELECT my_column FROM my_schema.my_table;  
SELECT "my_column" FROM "my_schema"."my_table";  
SELECT my_schema.my_function(my_column) from my_schema.my_table;
```

Reserved words

A series of reserved words exist, which cannot be used as regular identifiers. By way of example, the keyword, SELECT, is a reserved word. If a table needs to be created with the name, SELECT, this will only be possible if the name is in double quotation marks. "SELECT" as a table name is, therefore, different to a table name such as "Select" or "seLect".

The list of reserved words in Exasol can be found in the [EXA_SQL_KEYWORDS](#) system table.

2.1.3. Regular expressions

Regular expressions can be used to filter or replace strings by certain patterns. Exasol supports the **PCRE** dialect which has a powerful syntax and exceeds the functionality of dialects like **POSIX BRE** or **POSIX ERE**.

This chapter shall describe the basic functionality of regular expressions in Exasol. Detailed information about the **PCRE** dialect can be found on www.pcre.org [http://www.pcre.org].

Regular expressions can be used in the following functions and predicates:

- [REGEXP_INSTR](#)
- [REGEXP_REPLACE](#)
- [REGEXP_SUBSTR](#)
- [\[NOT\] REGEXP_LIKE](#)

Examples

In the description of the corresponding scalar functions and predicates you can find examples of regular expressions within SQL statements. The following examples shall demonstrate the general possibilities of regular expressions:

Meaning	Regular Expression ^a
American Express credit card number (15 digits and starting with 34 or 37)	3[47][0-9]{13}
IP addresses like 192.168.0.1	\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
Floating point numbers like -1.23E-6 or 40	[+-]?[0-9]*\.[0-9]+([+-]?[eE][0-9]+)?
Date values with format YYYY-MM-DD, e.g. 2010-12-01 (restricted to 1900-2099)	(19 20)\d\d-(0[1-9] 1[012])-(0[1-9] 12)[0-9]3[01])
Email addresses like hello.world@yahoo.com	(?i)[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}

^aPlease note that the example expressions are simplified for clearness reasons. Productive expressions for e.g. identifying valid email addresses are much more complex.

Pattern elements

The following elements can be used in regular expressions:

Table 2.1. Pattern elements in regular expressions

Element	Meaning within the regular expression
General elements	
.	Matches any single character except a newline - if modifier (?s) is set, then newlines are matched, too.
^	Begin of a string - if modifier (?m) is set, also the begin of a line is matched (after a newline)
\$	End of a string - if modifier (?m) is set, also the end of a line is matched (before a newline)
(?#)	Comment which is not evaluated (e.g. (?#This is a comment))
Quantifiers and Operators	
{}	Minimum/Maximum quantifier (e.g. {1,5} for 1-5 occurrences or {3} for exactly three occurrences or {,7} for at maximum 7 occurrences)
{ }?	similar to {}, but non-greedy (i.e. the minimal number of occurrences is used)
?	0 or 1 quantifier (<i>at most 1 time</i>) - similar to {0,1}
*	0 or more quantifier (<i>any</i>) - similar to {0,}
*?	Similar to *, but non-greedy (i.e. the minimal number of occurrences is used)
+	1 or more quantifier (<i>at least 1 time</i>) - similar to {1,}
+?	Similar to +, but non-greedy (i.e. the minimal number of occurrences is used)
	Alternative operator (e.g. a b c for 'a', 'b' or 'c'). Please consider that the first matching alternative is used. Hence the order of the alternatives can be relevant.
()	Definition of a subpattern (for grouping, e.g. (abc)+ for at least 1 time the word 'abc'). Furthermore, Captures are defined by the brackets (see below).
(?(?=pattern1)pattern2 pattern3)	Depending whether the pattern in the first brackets matches, the following pattern is evaluated (e.g. (?(?=0-9)[0-9]* [a-z]*) searches only for letters from a-z, if the first character is no digit).
Characters classes via []	
[]	Definition of a character class (e.g. [abc] for 'a', 'b', or 'c')
\	Escape character within a character class (e.g. [a\\]) for 'a' or '\\')
-	Defines a range (e.g. [0-9] for digits)
^	Negation of a character class (e.g. [^0-9] for non-digits)
Usage of backslash (\)	
\	Escape character to be able to use special characters (e.g. \+ for character '+')
\R	Any newline character (newline or carriage return)
\n	Newline
\r	Carriage return
\f	Form feed
\t	Tab
\e	Escape
\d	Any digit (0-9)
\D	Any non-digit
\w	Any word character (alphanumeric characters and _)
\W	Any non-word character
\s	Any whitespace character
\S	Any character except a whitespace character

Element	Meaning within the regular expression
Modifiers (set/unset options beginning from the current position)	
(?i)	Case insensitive
(?m)	^ and \$ also consider newlines
(?s)	. also matches newlines
(?x)	Ignore whitespace characters completely
(?U)	Non-greedy search (tries to find the shortest possible match)
(?X)	Any backslash that is followed by a letter that has no special meaning causes an error. E.g. \a is not allowed. Without this modifier, \a is evaluated as 'a'.
(?ims)	Switch on multiple modifiers (<i>i</i> , <i>m</i> and <i>s</i> are switched on)
(?i-ms)	Switch off modifiers (<i>i</i> is switched on, <i>m</i> and <i>s</i> are switched off)
(*CR)	By the use of those three modifiers you can specify which newline characters are considered by the special characters ^, \$ and .
(*LF)	(*CR) Only carriage return (\r) matches as newline character
(*CRLF)	(*LF) Only newline (\n) matches as newline character (*CRLF) Both alternatives match as newline characters
Example: pattern (*LF) (?ms)^abc . def\$ matches the string 'abc\ndef'.	
Captures	
\1	Back-References from the first up to the ninth capture.
\2	Captures are defined by subpatterns which are delimited by brackets. They are numbered by the opening brackets, from left to right. If you want to use more than 9 captures, you have to name a capture (see below).
...	
\9	Example: pattern (a(b c))([0-9] *) applied on the string 'ab123' results in the following captures: \1 ab \2 b \3 123
\0	This special capture always references the whole string.
(?P<name>)	With this notation you can name a capture, which should be used for clarity if you want to use many captures or brackets. Afterwards, you can reference the capture via \g<name>. Important: name must start with a non-digit. Named captures doesn't affect the ascending numeration (see example). Example: pattern (?P<all>([a-zA-Z]*) (?P<digits>[0-9] *)) applied on the string 'user123' results in the following captures: \1 and \g<all> user123 \2 user \3 and \g<digits> 123

Element	Meaning within the regular expression
(?:)	<p>Such brackets are not counted for captures.</p> <p>Example: pattern <code>(a(?:b c))([0-9])*</code> applied on the string 'ab123' results in the following captures:</p> <p>\1 ab \2 123</p>
(?:i:)	To simplify the syntax, you can set/unset modifiers between ? and :

2.2. SQL statements

2.2.1. Definition of the database (DDL)

The structure of the database is defined by means of the Data Definition Language (DDL). This concerns, in particular, the creation of schemas and all of the schema objects contained therein such as tables, views, functions, and scripts.

CREATE SCHEMA

Purpose

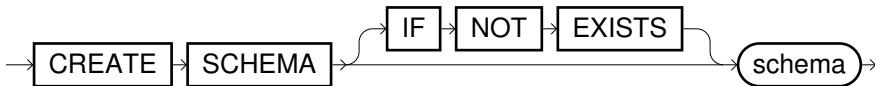
A new schema is created with this statement. This can either be a physical schema where you can create standard schema objects or a virtual schema which is some kind of virtual link to an external data source, mapping the remote tables including its data and meta data into the virtual schema.

Prerequisite(s)

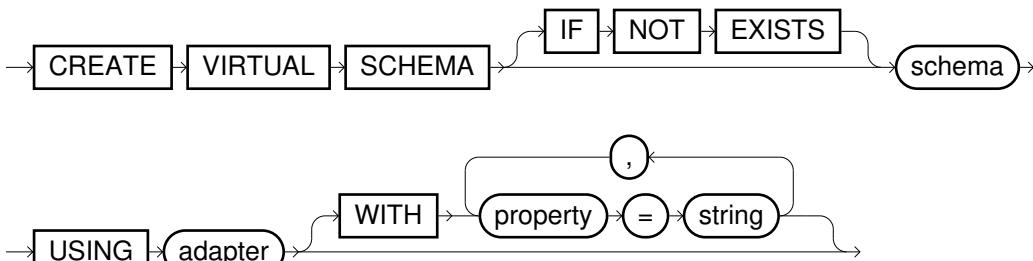
- System privilege CREATE SCHEMA or CREATE VIRTUAL SCHEMA
- For virtual schemas, you also need the EXECUTE privilege for the corresponding adapter script.

Syntax

create_schema ::=



create_virtual_schema ::=



Note(s)

- A user can create more than one schema.
- The schema that has been created belongs to the current user, however, it can be assigned to another user or role with the [ALTER SCHEMA](#) statement.
- When creating a schema one changes to the new schema. This means that it is set to the CURRENT_SCHEMA, above all this affects the name resolution.
- If you have specified the option IF NOT EXISTS, no error message will be thrown if the schema already exists. Furthermore, you change to this schema.
- Details about the concept of virtual schemas can be found in [Section 3.7, “Virtual schemas”](#). The USING option specifies the adapter udf script which defines the content of the virtual schema. Via the WITH clause you can specify certain properties which will be used by the adapter script.



Please note that virtual schemas are part of the Advanced Edition of Exasol.

Example(s)

```
CREATE SCHEMA my_schema;
CREATE VIRTUAL SCHEMA hive
  USING adapter.jdbc_adapter
  WITH
    SQL_DIALECT      = 'HIVE'
    CONNECTION_STRING = 'jdbc:hive2://localhost:10000/default'
    SCHEMA_NAME       = 'default'
    USERNAME          = 'hive-usr'
    PASSWORD          = 'hive-pwd';
```

DROP SCHEMA

Purpose

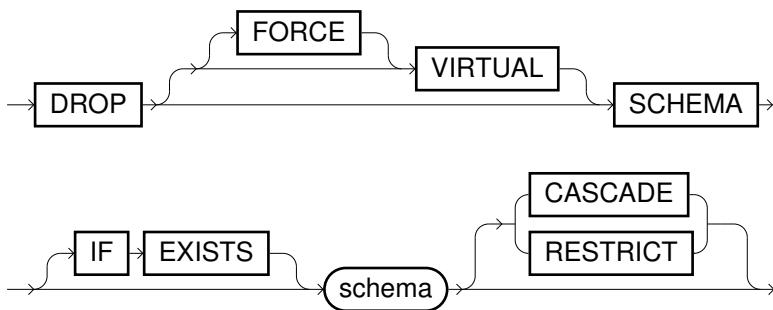
This statement can be used to completely delete a schema and all of the objects contained therein.

Prerequisite(s)

- Physical schema: System privilege `DROP ANY SCHEMA` or the schema belongs to the current user or one of that users roles.
- Virtual schema: System privilege `DROP ANY VIRTUAL SCHEMA` or the schema belongs to the current user or one of that users roles.

Syntax

`drop_schema ::=`



Note(s)

- `RESTRICT`: only deletes the schema if it is empty.
- `CASCADE`: deletes the schema and all of the schema objects contained therein. Furthermore, all foreign keys are deleted which reference one of the tables in that schema.
- If neither `CASCADE` nor `RESTRICT` is specified, `RESTRICT` is applied.
- If the optional `IF EXISTS` clause is specified, then the statement does not throw an exception if the schema does not exist.
- A user can own more than one schema.

- Details about the concept of virtual schemas can be found in [Section 3.7, “Virtual schemas”](#). If you specify the FORCE option, then the corresponding adapter script is not informed by that action. Otherwise, the adapter script will be called and could react to that action depending on its implementation.



Please note that virtual schemas are part of the Advanced Edition of Exasol.

Example(s)

```
DROP SCHEMA my_schema;
DROP SCHEMA my_schema CASCADE;
DROP VIRTUAL SCHEMA my_virtual_schema;
```

ALTER SCHEMA

Purpose

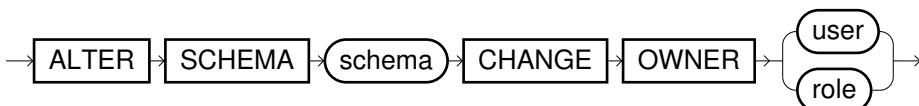
This statement alters a schema. With the CHANGE OWNER option you can assign a schema to another user or role. The schema and all the schema objects contained therein then belong to the specified user or the specified role. And for virtual schemas, you can change its properties or refresh its metadata.

Prerequisite(s)

- Physical schemas: The system privilege ALTER ANY SCHEMA or object privilege ALTER on the schema. Simply owning the schema is insufficient!
- Virtual schemas:
 - CHANGE OWNER: The system privilege ALTER ANY VIRTUAL SCHEMA or object privilege ALTER on the schema. Simply owning the schema is insufficient!
 - SET: System privilege ALTER ANY VIRTUAL SCHEMA, object privilege ALTER on s or the schema is owned by the current user or one of its roles. Additionally, access rights are necessary on the corresponding adapter script.
 - REFRESH: System privileges ALTER ANY VIRTUAL SCHEMA or ALTER ANY VIRTUAL SCHEMA REFRESH, object privilege ALTER or REFRESH on s, or s is owned by the current user or one of its roles. Additionally, access rights are necessary on the corresponding adapter script.

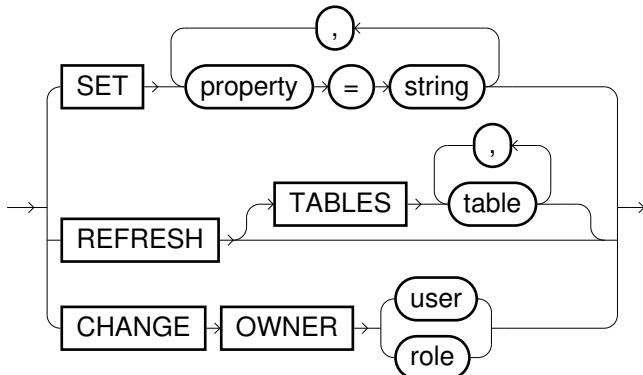
Syntax

alter_schema ::=



alter_virtual_schema ::=





Note(s)

- This command is not part of the SQL standard.
- Details about the concept of virtual schemas can be found in [Section 3.7, “Virtual schemas”](#). The REFRESH triggers an update of the metadata of a virtual schema, i.e. the underlying objects of the external data source. This update is only done explicitly by that statement and not implicitly when querying virtual schemas, because otherwise transaction conflicts would be possible for read-only transactions.



Please note that virtual schemas are part of the Advanced Edition of Exasol.

Example(s)

```

ALTER SCHEMA s1 CHANGE OWNER user1;
ALTER SCHEMA s1 CHANGE OWNER role1;
ALTER VIRTUAL SCHEMA s2
  SET CONNECTION_STRING = 'jdbc:hive2://localhost:10000/default';
ALTER VIRTUAL SCHEMA s2 REFRESH;
  
```

CREATE TABLE

Purpose

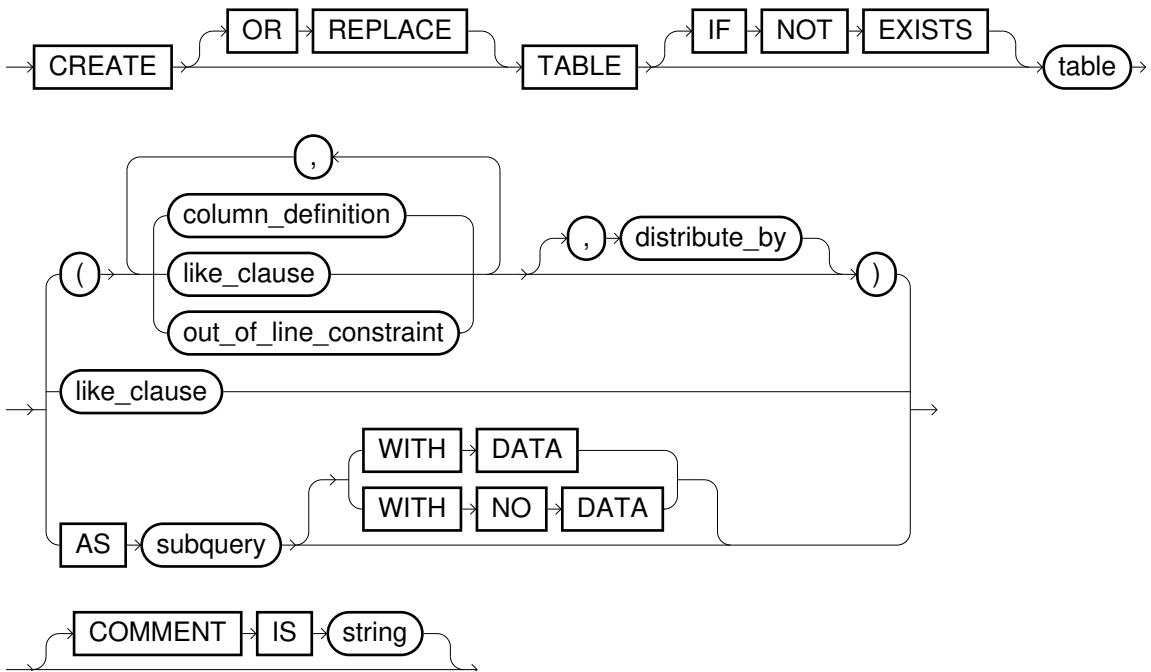
This statement can be used to create a table. This occurs by specifying the column types or other tables as sample or assigning a subquery.

Prerequisite(s)

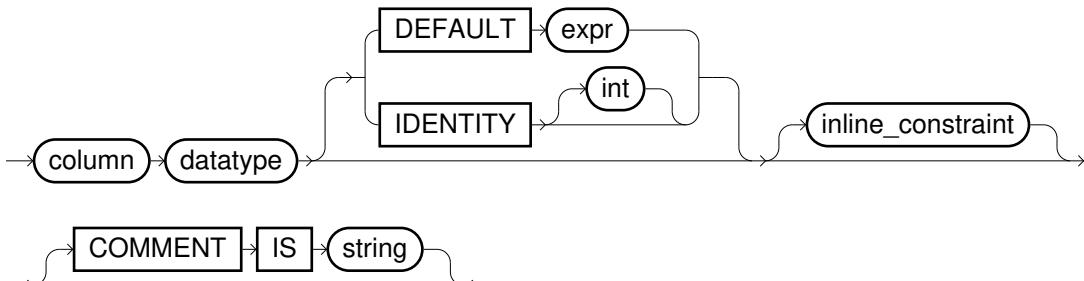
- System privilege `CREATE TABLE` system privilege if the table shall be created in your own schema or in that of an assigned role or `CREATE ANY TABLE`. If the table is defined via a subquery, the user requires the appropriate `SELECT` privileges for all the objects referenced in the subquery.
- If the `OR REPLACE` option has been specified and the table already exists, the rights for `DROP TABLE` are also needed.

Syntax

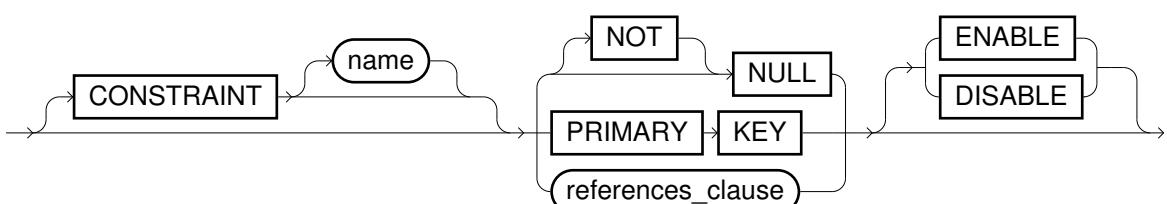
`create_table ::=`



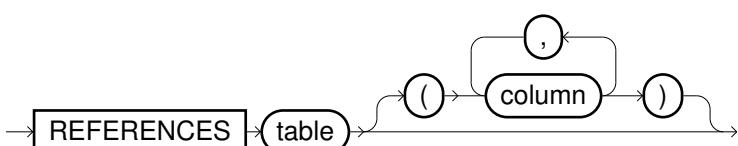
column_definition::=



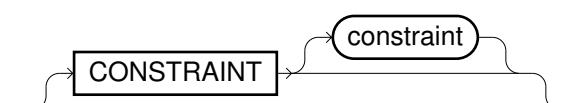
inline_constraint::=

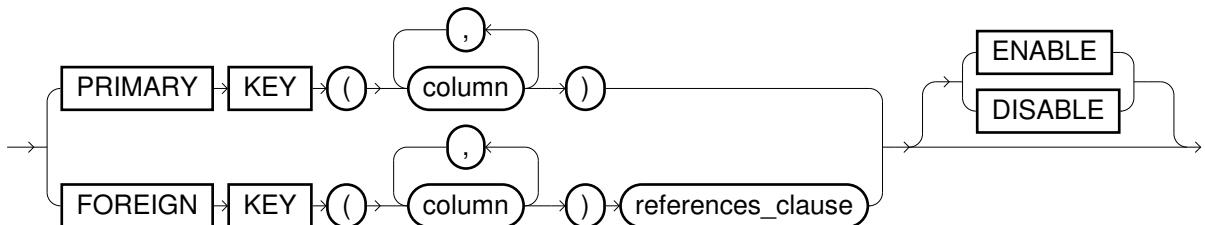


references_clause::=

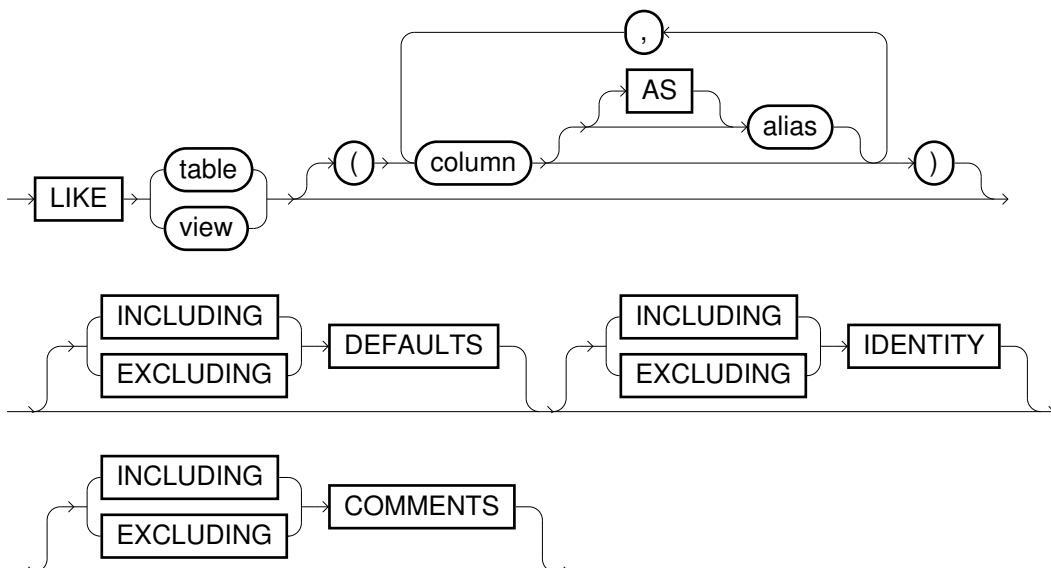


out_of_line_constraint::=

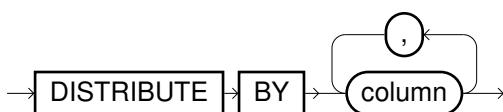




like_clause ::=



distribute by::=



Note(s)

- The OR REPLACE option can be used to replace an already existing table without having to explicitly delete this by means of [DROP TABLE](#).
 - If you have specified the option IF NOT EXISTS, no error message will be thrown if the table already exists.
 - Detailed information on the usable data types can be found in [Section 2.3, “Data types”](#).
 - For details about constraints see also command [ALTER TABLE \(constraints\)](#).
 - A specified default value must be convertible to the data type of its column. Detailed information on the permitted expressions for default values can be found in [Section 2.3.5, “Default values”](#).
 - Details on identity columns can be found in [Section 2.3.6, “Identity columns”](#).
 - If you use the WITH NO DATA option within the CREATE TABLE AS statement, the query is not executed, but an empty result table with same data types will be defined.
 - The following should be noted with regards to table definition with a (CREATE TABLE AS) subquery: if columns are not directly referenced in the subquery but assembled expressions, aliases must be specified for these columns.
 - Alternatively to the CREATE TABLE AS statement you can also the SELECT INTO TABLE syntax (see [SELECT INTO](#)).
 - Using the LIKE clause you can create an empty table with same data types like the defined sample table or view. If you specify the option INCLUDING DEFAULTS, INCLUDING IDENTITY or INCLUDING COMMENTS then all default values, its identity column and its column comments are adopted. A NOT NULL constraint is always adopted, in contrast to primary or foreign keys (never adopted).

 If the `like_clause`, but no `distribution_clause` is specified, then the distribution key is implicitly adopted from the derived tables if all distribution key columns of one table and no distribution key column of other tables were selected.

- `LIKE` expressions and normal column definition can be mixed, but have then to be enclosed in brackets.
- The `distribution_clause` defines the explicit distribution of the table across the cluster nodes. For details see also the notes of the command [ALTER TABLE \(distribution\)](#).
- Comments on table and columns can also be set or changed afterwards via the command [COMMENT](#).

Example(s)

```
CREATE TABLE t1 (a VARCHAR(20),
                 b DECIMAL(24,4) NOT NULL,
                 c DECIMAL DEFAULT 122,
                 d DOUBLE,
                 e TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                 f BOOL);
CREATE TABLE t2 AS SELECT * FROM t1;
CREATE OR REPLACE TABLE t2 AS SELECT a,b,c+1 AS c FROM t1;
CREATE TABLE t3 AS SELECT count(*) AS my_count FROM t1 WITH NO DATA;
CREATE TABLE t4 LIKE t1;
CREATE TABLE t5 (id int IDENTITY PRIMARY KEY,
                 LIKE t1 INCLUDING DEFAULTS,
                 g DOUBLE,
                 DISTRIBUTE BY a,b);
SELECT * INTO TABLE t6 FROM t1;
```

SELECT INTO

Purpose

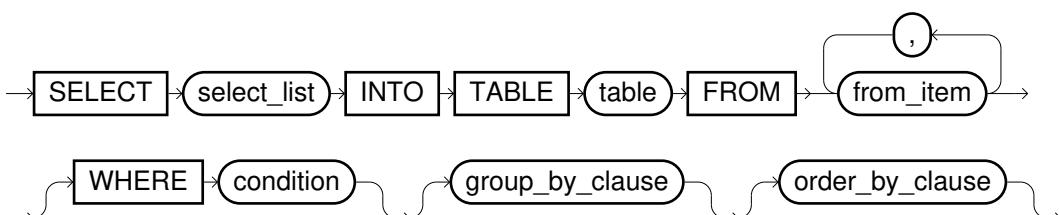
Via this command you can create a table, similar to `CREATE TABLE AS` (see [CREATE TABLE](#)).

Prerequisite(s)

- System privilege `CREATE TABLE` system privilege if the table shall be created in your own schema or in that of an assigned role or `CREATE ANY TABLE`. Furthermore, the user requires the appropriate `SELECT` privileges for all referenced objects.

Syntax

`select_into ::=`



Note(s)

- For details about the syntax elements (like e.g. `select_list`, ...) see [SELECT](#).

- If columns are not directly referenced in the select list, but compound expressions, aliases must be specified for these columns.

Example(s)

```
SELECT * INTO TABLE t2 FROM t1 ORDER BY 1;
```

DROP TABLE

Purpose

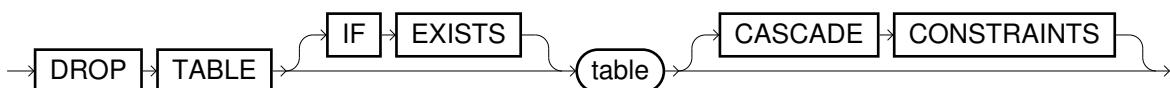
This statement can be used to delete a table.

Prerequisite(s)

- System privilege DROP ANY TABLE or the table belongs to the current user or one of the roles of the current user.

Syntax

drop_table ::=



Note(s)

- If a foreign key references the table which should be deleted, you have to specify the option CASCADE CONSTRAINTS. In this case the foreign key is also deleted even though the referencing table doesn't belong to the current user.
- If the optional IF EXISTS clause is specified, then the statement does not throw an exception if the table does not exist.

Example(s)

```
DROP TABLE my_table;
```

ALTER TABLE (column)

Purpose

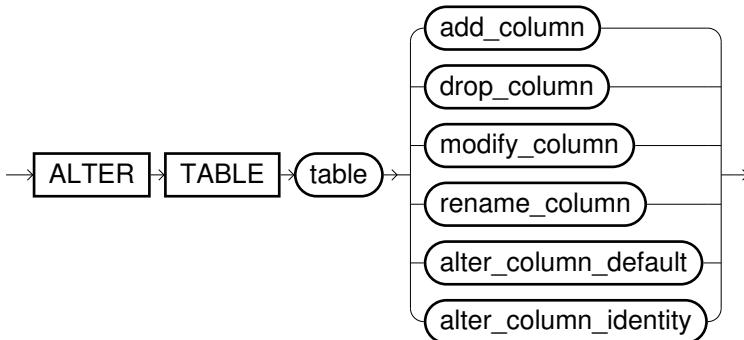
Add, drop, change datatype and rename a column or define default values and column identities.

Prerequisite(s)

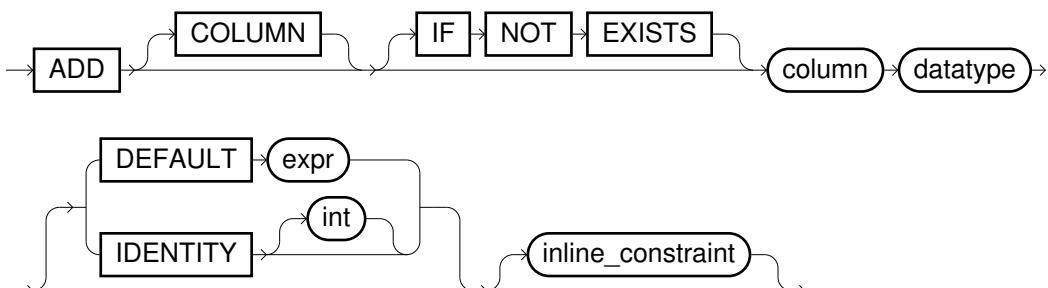
- System privilege ALTER ANY TABLE, object privilege ALTER on the table or its schema, or the table belongs to the current user or one of his roles.

Syntax

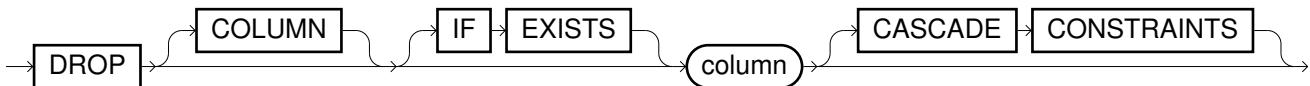
alter_column ::=



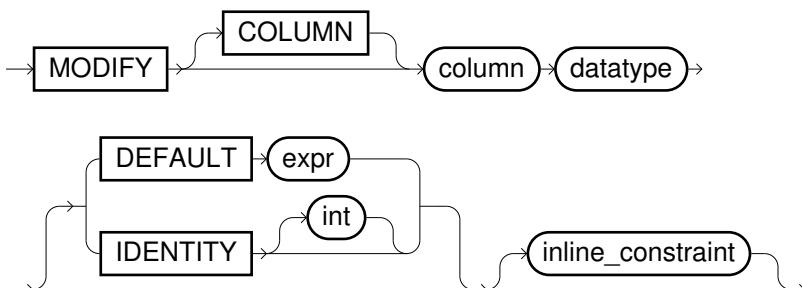
add_column ::=



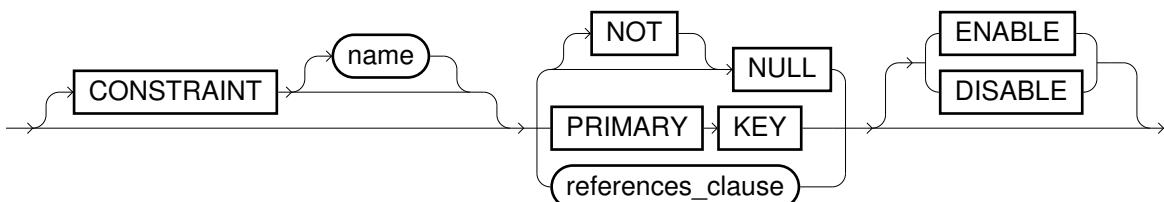
drop_column ::=



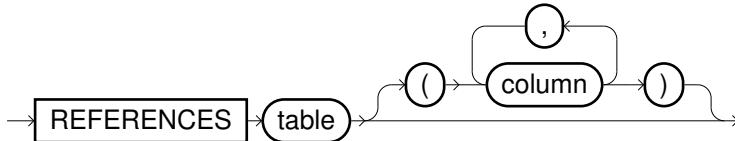
modify_column ::=



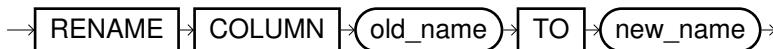
inline_constraint ::=



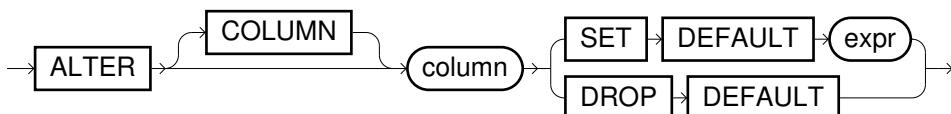
references_clause ::=



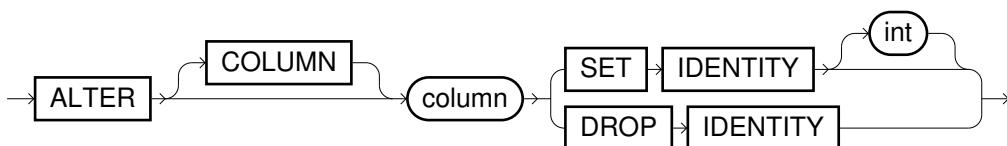
`rename_column ::=`



`alter_column_default ::=`



`alter_column_identity ::=`



Note(s)

ADD COLUMN

- If the table already contains rows, the content of the inserted column will be set to the default value if one has been specified, otherwise it will be initialized with NULL.
- If the default value `expr` is not appropriate for the specified data type, an error message is given and the statement is not executed. Details on the permissible expressions for the default value `expr` can be found in [Section 2.3.5, “Default values”](#).
- For identity columns a monotonically increasing number is generated if the table already possesses rows. Details on identity columns can be found in [Section 2.3.6, “Identity columns”](#).
- If you have specified the option IF NOT EXISTS, no error message will be thrown if the schema already exists.
- For details about constraints see also command [ALTER TABLE \(constraints\)](#).

DROP COLUMN

- If a foreign key references the column which should be deleted, you have to specify the option CASCADE CONSTRAINTS. In this case the foreign key is also deleted even though the referencing table doesn't belong to the current user.
- If a column is dropped which is part of the distribution key (see also [ALTER TABLE \(distribution\)](#)), then the distribution keys are deleted completely and the new rows of the table will be distributed randomly across the cluster nodes.
- If the optional IF EXISTS clause is specified, then the statement does not throw an exception if the column does not exist.

MODIFY COLUMN

- The specified types must be convertible. This concerns the data types (e.g. `BOOL` is not convertible into `TIMESTAMP`) and the content of the column (`CHAR(3)` with content '123' can be converted into `DECIMAL`, but 'ABC' can not).

- A default value must be convertible to the data type. If a default value has not been specified, any older default value that exists will be used; this must be appropriate for the new data type. Details on the permissible expressions for the default value *expr* can be found in [Section 2.3.5, “Default values”](#).
- Details on identity columns can be found in [Section 2.3.6, “Identity columns”](#). If the identity column option has not been specified for an existing identity column, then the column still has the identity column property and the new data type must be appropriate.
- For details about constraints see also command [ALTER TABLE \(constraints\)](#). By specifying constraints in the MODIFY command you can only add constraints, but not modify them. Exception: If you specify a NULL constraint, a previously existing NOT NULL constraint is removed. If you don't specify a constraint, then existing constraints are not changed.
- If a column is modified which is part of the distribution key (see also [ALTER TABLE \(distribution\)](#)), the table is redistributed.

RENAME COLUMN

- This statement will not change the contents of the table.
- The table may not contain a column with the name *new_name*.

ALTER COLUMN DEFAULT

- This statement will not change the contents of the table.
- A default value must be convertible to the data type.
- Details on the permissible expressions for the default value *expr* can be found in [Section 2.3.5, “Default values”](#).
- DROP DEFAULT should be given priority over SET DEFAULT NULL.

ALTER COLUMN IDENTITY

- The content of the tables is not affected by this statement.
- When specifying the optional parameter *int* the number generator will be set to this number, otherwise to 0. Starting with this number monotonically increasing numbers are generated for INSERT statements which do not insert an explicit value for the identity column. Please consider that those numbers will be generated monotonically increasing, but can include gaps. By the use of this command you can also "reset" the number generator.
- Details on identity columns can be found in [Section 2.3.6, “Identity columns”](#).

Example(s)

```

ALTER TABLE t ADD COLUMN IF NOT EXISTS new_dec DECIMAL(18,0);
ALTER TABLE t ADD new_char CHAR(10) DEFAULT 'some text';

ALTER TABLE t DROP COLUMN i;
ALTER TABLE t DROP j;

ALTER TABLE t MODIFY (i DECIMAL(10,2));
ALTER TABLE t MODIFY (j VARCHAR(5) DEFAULT 'text');
ALTER TABLE t MODIFY (k INTEGER IDENTITY(1000));

ALTER TABLE t RENAME COLUMN i TO j;

ALTER TABLE t ALTER COLUMN v SET DEFAULT CURRENT_USER;
ALTER TABLE t ALTER COLUMN v DROP DEFAULT;

ALTER TABLE t ALTER COLUMN id SET IDENTITY(1000);
ALTER TABLE t ALTER COLUMN id DROP IDENTITY;

```

ALTER TABLE (distribution)

Purpose

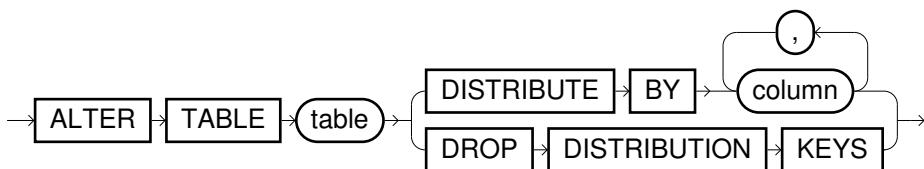
By using this statement you can control the distribution of rows across the cluster nodes.

Prerequisite(s)

- System privilege ALTER ANY TABLE, object privilege ALTER on the table or its schema, or the table belongs to the current user or one of his roles.

Syntax

distribute_by::=



Note(s)

- Explicitly distributed tables can accelerate certain SQL significantly. E.g. a join between two tables which are **both** distributed by the join-columns can be executed locally on the cluster-nodes. GROUP BY queries on single tables can also be accelerated if all distribution keys (columns) are part of the grouping elements. However it is senseless to explicitly distribute small tables with only a few thousands rows.



The execution time of this command can take some time. It should only be used if you are aware of its consequences. In particular you can influence the performance of queries negatively.

- If the table distribution was not defined explicitly, the system distributes the rows equally across the cluster nodes.
- If a table is explicitly distributed, the distribution of the rows across the cluster nodes is determined by the hash value of the distribution keys. The order of the defined columns is irrelevant. The distribution of a table is sustained, especially when inserting, deleting or updating rows.
- The actual mapping between values and database nodes can be evaluated using the function [VALUE2PROC](#).
- If you define inappropriate distribution keys which would lead to a heavy imbalance of rows in the cluster, the command is aborted and a corresponding error message is thrown.
- The explicit distribution of a table can also be defined directly in the [CREATE TABLE](#) statement.
- By setting or unsetting the distribution key, all internal indices are recreated.
- Via the [DROP DISTRIBUTION KEYS](#) statement you can undo the explicit distribution of a table. Afterwards the rows will again be distributed randomly, but equally across the cluster nodes.
- By executing the [DESCRIBE](#) statement or by using the information in the system table [EXA_ALL_COLUMNS](#) you can identify the distribution keys of a table. In system table [EXA_ALL_TABLES](#) you can see whether a table is explicitly distributed.

Example(s)

```

CREATE TABLE my_table (i DECIMAL(18,0), vc VARCHAR(100), c CHAR(10));
ALTER TABLE my_table DISTRIBUTED BY i, c;

DESCRIBE my_table;
  
```

COLUMN_NAME	SQL_TYPE	NULLABLE	DISTRIBUTION_KEY
I	DECIMAL(18, 0)	TRUE	TRUE
VC	VARCHAR(100) UTF8	TRUE	FALSE
C	CHAR(10) UTF8	TRUE	TRUE

```
ALTER TABLE my_table DROP DISTRIBUTION KEYS;
```

ALTER TABLE (constraints)

Purpose

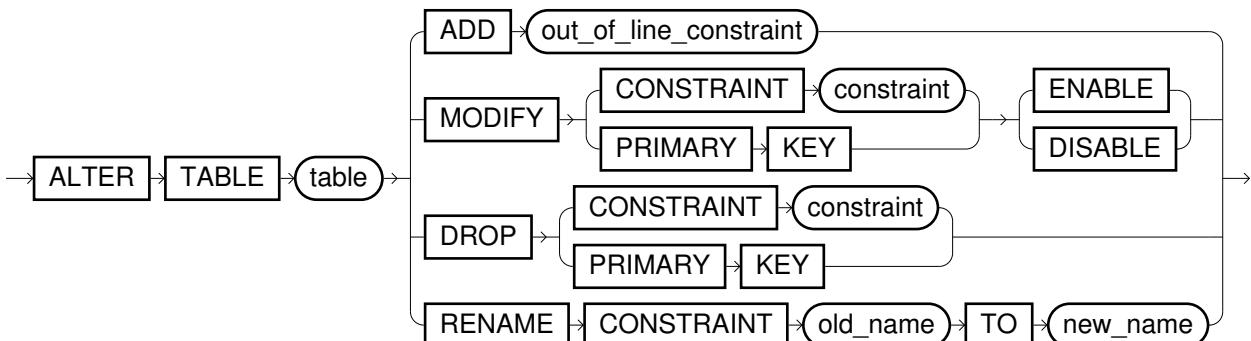
By using this command you can define, delete or modify constraints.

Prerequisite(s)

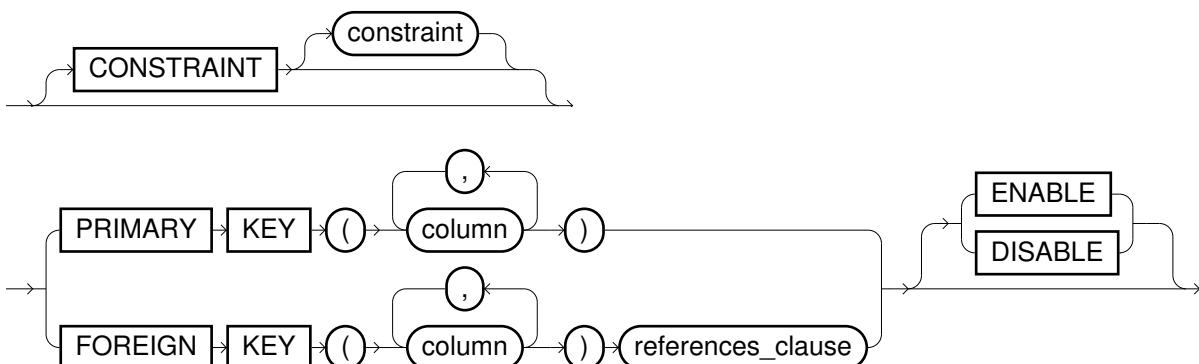
- System privilege ALTER ANY TABLE, object privilege ALTER on the table or its schema, or the table belongs to the current user or one of his roles.

Syntax

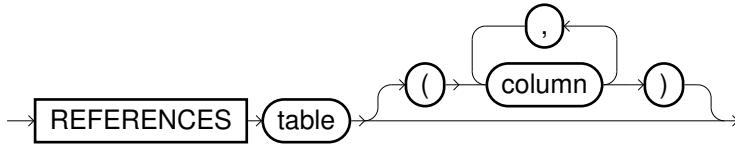
alter_table_constraint ::=



out_of_line_constraint ::=



references_clause ::=

**Note(s)**

- The following constraints can be defined for columns:

PRIMARY KEY All values have to be unique, NULL values are not allowed. A table may only have one primary key.

FOREIGN KEY A foreign key always references the primary key of a second table. The column content must either exist in the primary key column or must be NULL (in case of a composite key at least one of the columns). The datatype of a foreign key and its corresponding primary key must be identical. Foreign keys cannot reference a virtual object.

NOT NULL No NULL values can be inserted. A NOT NULL constraint can only be specified either directly in the table definition or via the ALTER TABLE MODIFY COLUMN statement.

- Constraints can have a name for easier identification and always have one of the following two states:

ENABLE The constraint is directly checked after DML statements (see [Section 2.2.2, “Manipulation of the database \(DML\)”,](#)). This process costs some time, but ensures the data integrity.

DISABLE The constraint is deactivated and not checked. This state can be interesting if you want to define the metadata within the database, but avoid a negative performance impact.

If no explicit state is defined, then the session parameter CONSTRAINT_STATE_DEFAULT is used (see also [ALTER SESSION](#)).

- The corresponding metadata for primary and foreign key constraints can be found in system tables [EXA_USER_CONSTRAINTS](#), [EXA_ALL_CONSTRAINTS](#) and [EXA_DBAA_CONSTRAINTS](#), the metadata for NOT NULL constraints in [EXA_USER_CONSTRAINT_COLUMNS](#), [EXA_ALL_CONSTRAINT_COLUMNS](#) and [EXA_DBAA_CONSTRAINT_COLUMNS](#). If no explicit name was specified, the system generates implicitly a unique name.
- Constraints can also be defined directly within the [CREATE TABLE](#) statement and can be modified by the command [ALTER TABLE \(constraints\)](#).

Example(s)

```

ALTER TABLE t1 ADD CONSTRAINT my_primary_key PRIMARY KEY (a);
ALTER TABLE t2 ADD CONSTRAINT my_foreign_key FOREIGN KEY (x) REFERENCES t1;
ALTER TABLE t2 MODIFY CONSTRAINT my_foreign_key DISABLE;
ALTER TABLE t2 RENAME CONSTRAINT my_foreign_key TO my_fk;
ALTER TABLE t2 DROP CONSTRAINT my_fk;

```

CREATE VIEW**Purpose**

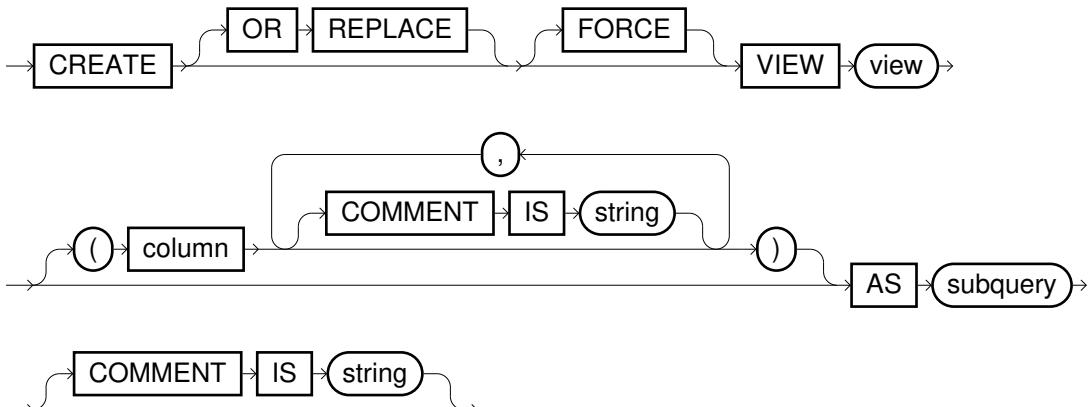
This statement can be used to create a view.

Prerequisite(s)

- System privilege CREATE VIEW if the view shall be created in your own schema or in that of an assigned role or CREATE ANY VIEW. Additionally, the owner of the view (who is not automatically the creator) must possess the corresponding SELECT privileges on all the objects referenced in the subquery.
- If the OR REPLACE option has been specified and the view already exists, the user also needs the same rights as for [DROP VIEW](#).

Syntax

create_view ::=



Note(s)

- An already existing view can be replaced with the OR REPLACE option, without having to explicitly delete this with [DROP VIEW](#).
- By specifying the FORCE option you can create a view without compiling its text. This can be very useful if you want to create many views which depend on each other and which otherwise must be created in a certain order. Please consider that syntax or other errors will not be found until the the view is first used.
- The column names of the view are defined through the specification of column aliases. If these are not specified, the column names are derived from the subquery. If columns are not directly referenced in the subquery but assembled expressions, aliases must be specified for these either within the subquery or for the entire view.
- The view creation text is limited to 20.000 characters.
- A view is set to INVALID If one of its underlying objects was changed (see also column STATUS in system tables like e.g. [EXA_ALL_VIEWS](#)). At the next read access, the system tries to automatically recompile the view. If this is successful, the view is valid again. Otherwise an appropriate error message is thrown. This means in particular that an invalid view can still be usable.

Example(s)

```

CREATE VIEW my_view as select x from t;
CREATE OR REPLACE VIEW my_view as select y from t;
CREATE OR REPLACE VIEW my_view (col_1) as select max(y) from t;

```

DROP VIEW

Purpose

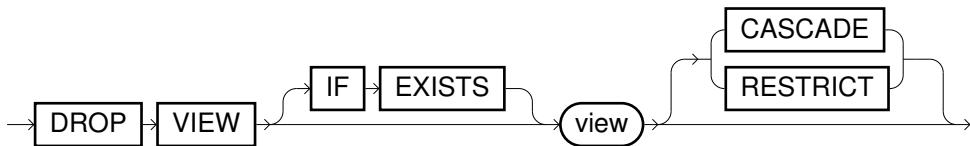
This statement can be used to delete a view.

Prerequisite(s)

- System privilege DROP ANY VIEW, or the view belongs to the current user or one of his roles.

Syntax

drop_view ::=

**Note(s)**

- `RESTRICT` or `CASCADE` are not of significance but are supported syntactically for reasons of compatibility.
- If the optional `IF EXISTS` clause is specified, then the statement does not throw an exception if the view does not exist.

Example(s)

```
DROP VIEW my_view;
```

CREATE FUNCTION**Purpose**

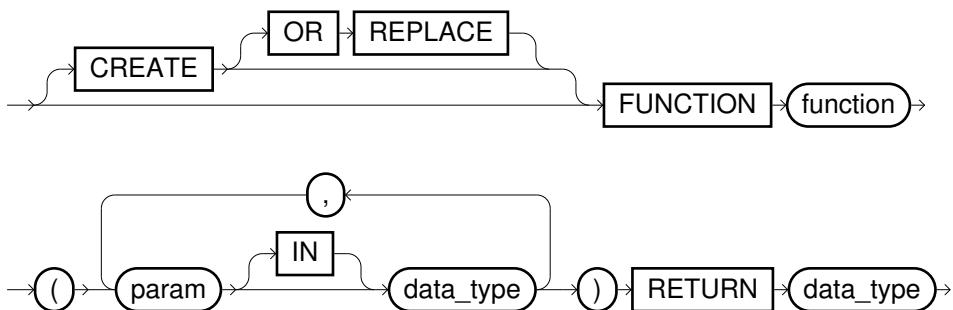
This statement can be used to create a user-defined function.

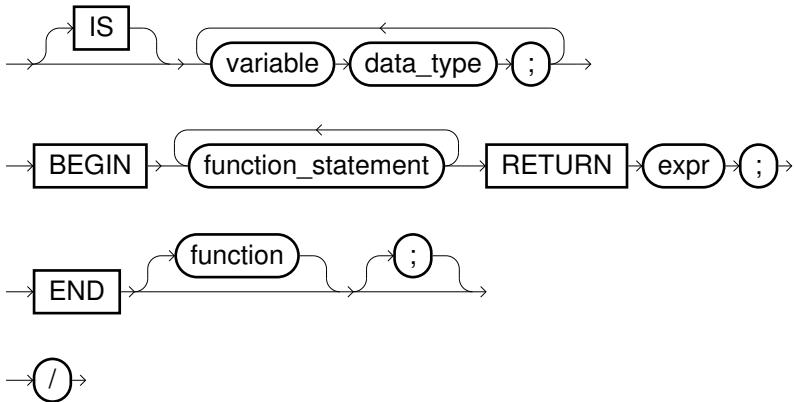
Prerequisite(s)

- System privilege `CREATE FUNCTION` if the function shall be created in your own schema or in that of an assigned role or `CREATE ANY FUNCTION`.
- If the `OR REPLACE` option has been specified and the function already exists, the rights for `DROP FUNCTION` are also needed.
- In order to use a function, one needs the system privilege `EXECUTE ANY FUNCTION`, the object privilege `EXECUTE` on the function or its schema, or must be the owner.

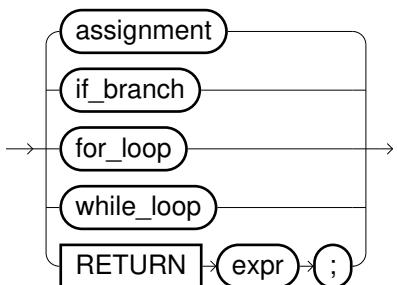
Syntax

create_function ::=

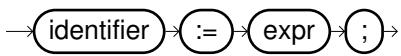




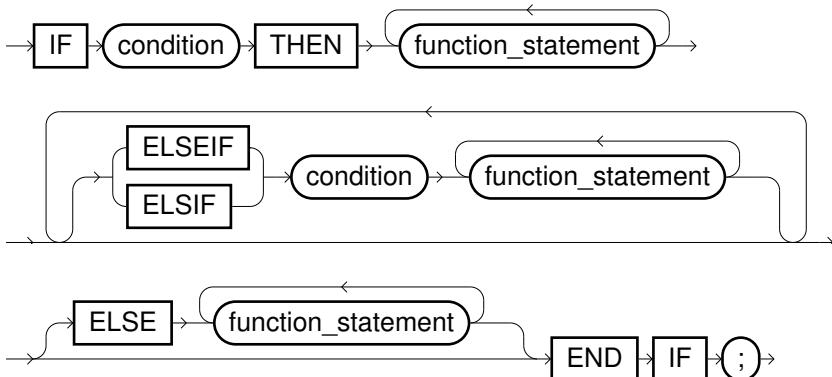
function_statement ::=



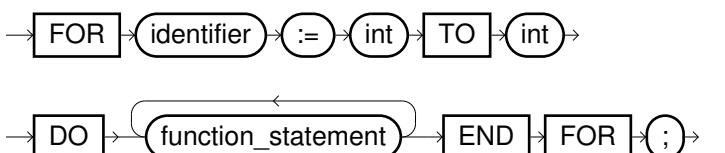
assignment ::=



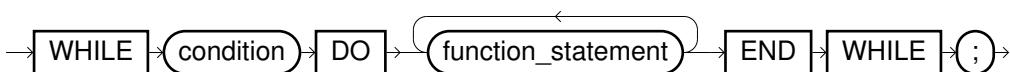
if_branch ::=



for_loop ::=



while_loop ::=



Note(s)

- The ending slash ('/') is only required when using EXAplus.
- The OR REPLACE option can be used to replace an already existing function without having to explicitly delete this with [DROP FUNCTION](#).
- The normal SQL data types are valid for variable declarations (see also [Section 2.3, “Data types”](#)).
- Variable assignments cannot be performed in the variable declaration, but only in the function body.
- Any scalar SQL expressions can be used for expressions (expr) e.g. all built-in (see [Section 2.9.1, “Scalar functions”](#)) and previously defined scalar functions are available .
- The counter variable of the FOR statement iterates over all integers which are defined by the border parameters. The number of iterations is evaluated as a constant before entering the FOR body and is never re-evaluated.
- Similar to normal SQL statements any number of comments can be inserted within a function definition (see also [Section 2.1.1, “Comments in SQL”](#)).
- The definition text of a user defined function can be found in the [EXA_ALL_FUNCTIONS](#) system table, etc. (see [Appendix A, System tables](#)).
- Scalar subqueries within functions cannot be parametrized. That means they cannot contain variables or parameters.

Example(s)

```

CREATE OR REPLACE FUNCTION percentage (fraction DECIMAL,
                                      entirety DECIMAL)
                                     RETURN VARCHAR(10)
IS
    res DECIMAL;
BEGIN
    IF entirety = 0
        THEN res := NULL;
    ELSE
        res := (100*fraction)/entirety;
    END IF;
    RETURN res || ' %';
END percentage;
/
SELECT fraction, entirety, percentage(fraction,entirety) AS PERCENTAGE
FROM my_table;

FRACTION ENTIRETY PERCENTAGE
----- -----
      1          2        50 %
      1          3        33 %
      3         24       12 %

```

```

-- Examples for function statements
-- assignment
    res := CASE WHEN input_variable < 0 THEN 0 ELSE input_variable END;

-- if-branch
    IF input_variable = 0 THEN
        res := NULL;
    ELSE
        res := input_variable;
    END IF;

-- for loop
    FOR cnt := 1 TO input_variable

```

```

DO
    res := res*2;
END FOR;

-- while loop
WHILE cnt <= input_variable
DO
    res := res*2;
    cnt := cnt+1;
END WHILE;

```

DROP FUNCTION

Purpose

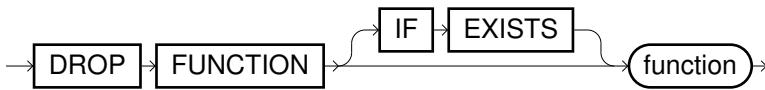
A user-defined function can be deleted with this statement.

Prerequisite(s)

- System privilege DROP ANY FUNCTION or the function belongs to the current user.

Syntax

drop_function ::=



Note(s)

- If the optional IF EXISTS clause is specified, then the statement does not throw an exception if the function does not exist.

Example(s)

```
DROP FUNCTION my_function;
```

CREATE SCRIPT

Purpose

A script can be created by this statement, either a user defined function (UDF), a scripting program or an adapter script.

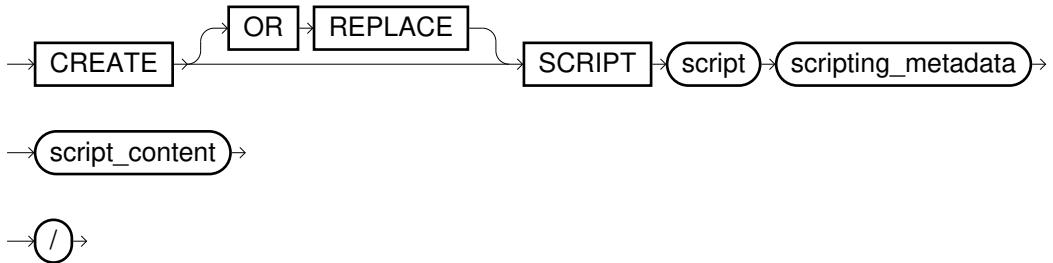
Prerequisite(s)

- System privilege CREATE SCRIPT if the script shall be created in your own schema or in that of an assigned role or CREATE ANY SCRIPT.
- If the OR REPLACE option has been specified and the script already exists, the rights for [DROP SCRIPT](#) are also needed.

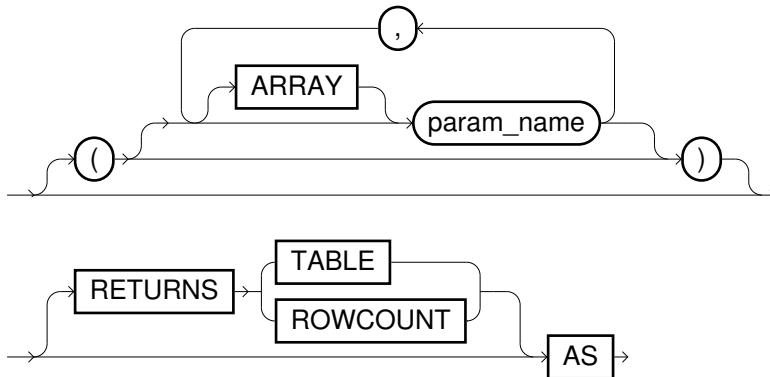
- To execute a script (either UDF or scripting program) you need the system privilege EXECUTE ANY SCRIPT, the object privilege EXECUTE on the script or its schema or the user has to be the owner of the script.

Syntax

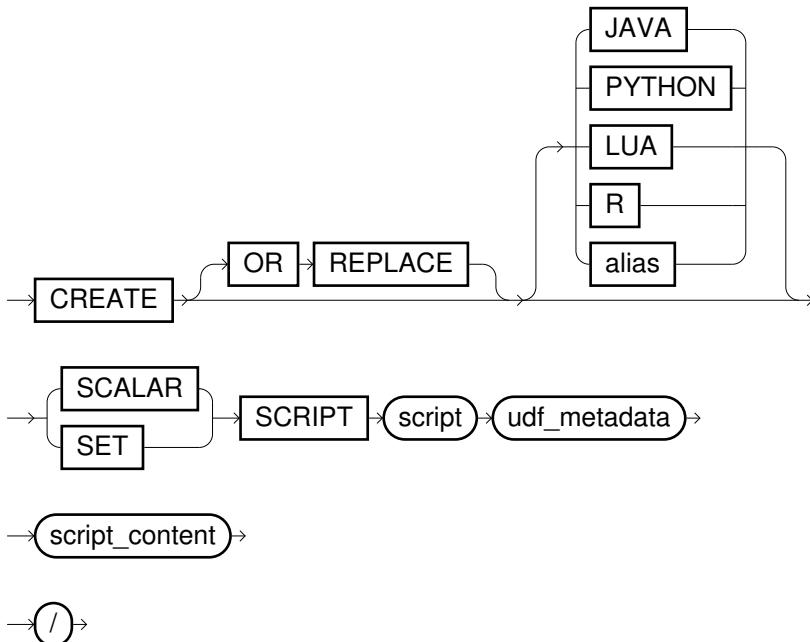
`create_scripting_script ::=`



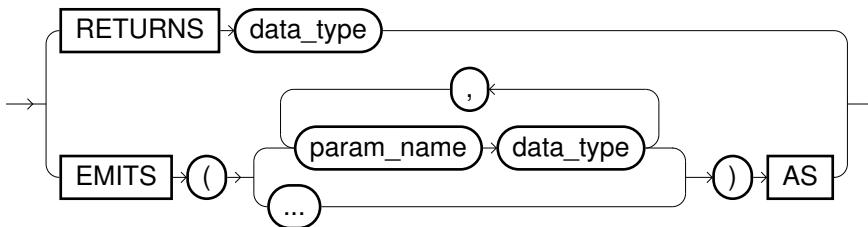
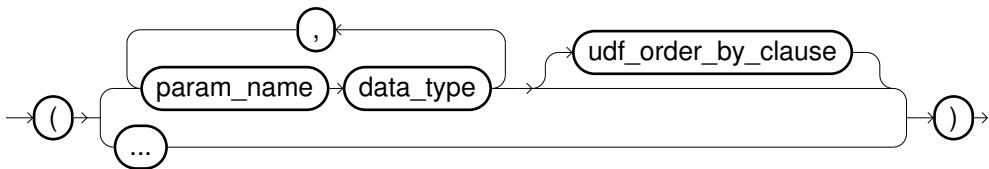
`scripting_metadata ::=`



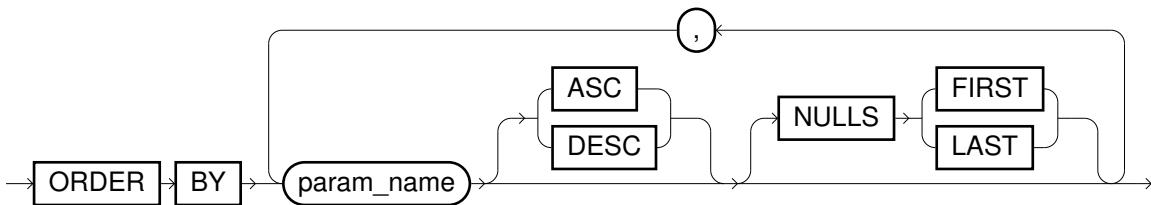
`create_udf_script ::=`



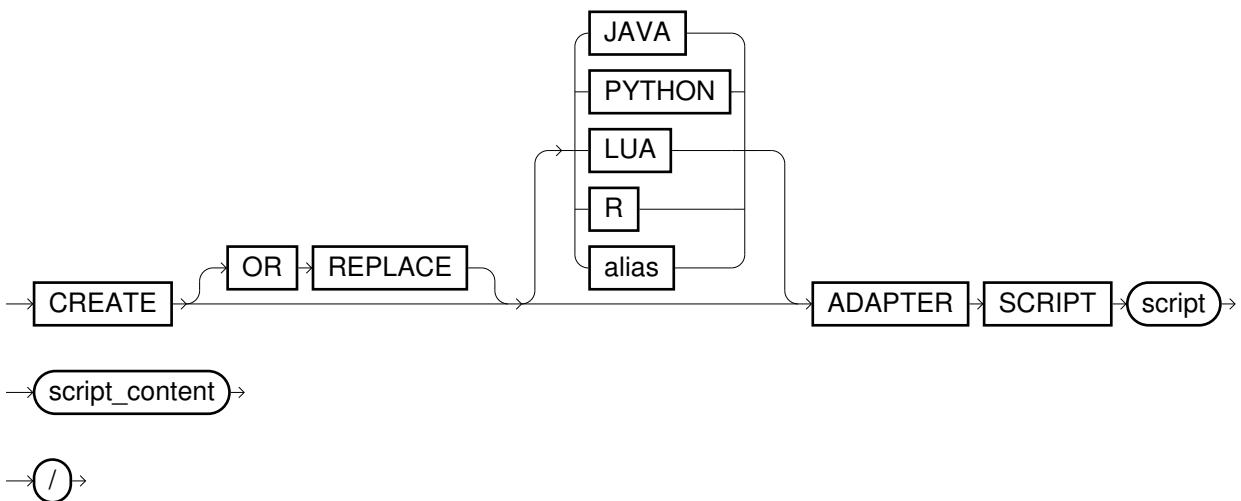
`udf_metadata ::=`



`udf_order_by_clause ::=`



`create_adapter_script ::=`



Note(s)

- An already existing script can be replaced with the `OR REPLACE` option, without having to explicitly delete this with [DROP SCRIPT](#).
- The ending slash ('/') is only required when using EXAplus.
- The Content of a script is integrated in the corresponding system tables (e.g. [EXA_ALL_SCRIPTS](#) - see also [Appendix A, System tables](#)). It begins from the first line after the `AS` keyword which is no blank line.
- If you want to integrate further script languages for user defined functions and adapter scripts, please see also [Section 3.6.5, “Expanding script languages using BucketFS”](#).
- Notes for scripting programs:
 - Scripting programs provide a way of controlling the execution of several SQL commands (e.g. for ETL jobs) and are executed with the [EXECUTE SCRIPT](#) command (see also [Section 3.5, “Scripting”](#)).

- For scripting programs only Lua is supported as programming language. More details can be found in [Section 3.5, “Scripting”](#).
- If neither of the two options RETURNS TABLE and RETURNS ROWCOUNT are specified, implicitly the option RETURNS ROWCOUNT is used (for details about those options see also section [Return value of a script](#)).
- You can execute a script via the statement [EXECUTE SCRIPT](#).
- Notes for user defined functions:
 - User defined functions ([UDF](#)) can be used directly within SELECT statements and can proceed big data volumes (see also [Section 3.6, “UDF scripts”](#)).



Please note that UDF scripts are part of the Advanced Edition of Exasol.

- Additionally to scalar functions, you can also create aggregate and analytical functions. Even MapReduce algorithms can be implemented. More details can be found in [Section 3.6, “UDF scripts”](#).
- If you define the ORDER BY clause, the groups of the SET input data are processed in an ordered way. You can also specify this clause in the function call within a SELECT statement.
- Notes for adapter scripts:
 - Details about adapter scripts and virtual schemas can be found in [Section 3.7, “Virtual schemas”](#).



Please note that virtual schemas are part of the Advanced Edition of Exasol.

- The existing open source adapters provided by Exasol can be found in our GitHub repository: <https://www.github.com/exasol>
- Adapter scripts can only be implemented in Java and Python.

Example(s)

```
-- define a reusable function definition
CREATE SCRIPT function_lib AS
    function min_max(a,b,c)
        local min,max
        if a>b then max,min=a,b
            else max,min=b,a
        end
        if c>max then max=c
        else if c<min then min=c
        end
        return min,max
    end
/
-- scripting program example for data insert
CREATE SCRIPT insert_low_high (param1, param2, param3) AS
    import('function_lib') -- accessing external function
    lowest, highest = function_lib.min_max(param1, param2, param3)
    query([[INSERT INTO t VALUES (:x, :y)]], {x=lowest, y=highest})
/
EXECUTE SCRIPT insert_low_high(1, 3, 4);
EXECUTE SCRIPT insert_low_high(2, 6, 4);
EXECUTE SCRIPT insert_low_high(3, 3, 3);

-- UDF example
CREATE LUA SCALAR SCRIPT my_average (a DOUBLE, b DOUBLE)
    RETURNS DOUBLE AS
function run(ctx)
```

```

if ctx.a == nil or ctx.b==nil
    then return NULL
end
return (ctx.a+ctx.b)/2
end
/

SELECT x,y,my_average(x,y) FROM t;

X          Y          MY_AVERAGE(T.X,T.Y)
-----
1           4           2.5
2           6           4
3           3           3

-- Adapter script example
CREATE JAVA ADAPTER SCRIPT my_script AS
  %jar hive_jdbc_adapter.jar
/

```

DROP SCRIPT

Purpose

A script can be dropped by this statement (UDF, scripting program or adapter script).

Prerequisite(s)

- System privilege DROP ANY SCRIPT or the current user is the owner of the script (i.e. the whole schema).

Syntax

drop_script::=



Note(s)

- If the optional IF EXISTS clause is specified, then the statement does not throw an exception if the script does not exist.
- In case of an adapter script still referenced by a virtual schema, an exception is thrown that the script cannot be dropped.

Example(s)

```
DROP SCRIPT my_script;
```

RENAME

Purpose

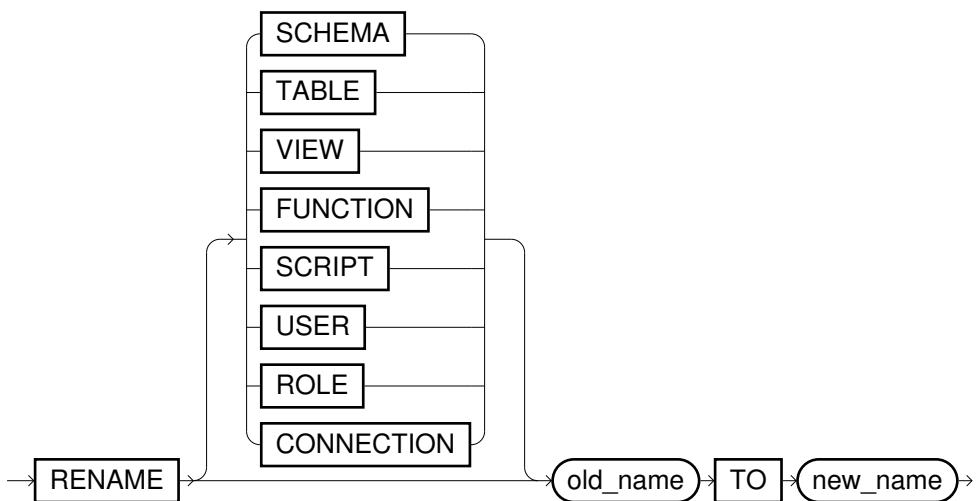
This statement makes it possible to rename schemas and schema objects.

Prerequisite(s)

- If the object is a schema, it must belong to the user or one of his roles.
- If the object is a schema object, the object must belong to the user or one of his roles (i.e. located in one's own schema or that of an assigned role).
- If the object is a user or role, the user requires the CREATE USER or CREATE ROLE.
- If the object is a connection, the user requires the ALTER ANY CONNECTION system privilege or he must have received the connection with the WITH ADMIN OPTION.

Syntax

rename ::=



Note(s)

- Schema objects cannot be shifted to another schema with the RENAME statement, i.e. "RENAME TABLE s1.t1 TO s2.t2" is not allowed.
- Distinguishing between schema, table, etc. is optional and only necessary if two identical schema objects share the same name.

Example(s)

```
RENAME SCHEMA s1 TO s2;  
RENAME TABLE t1 TO t2;  
RENAME s2.t3 TO t4;
```

COMMENT

Purpose

This command allows you to comment schemas and schema objects.

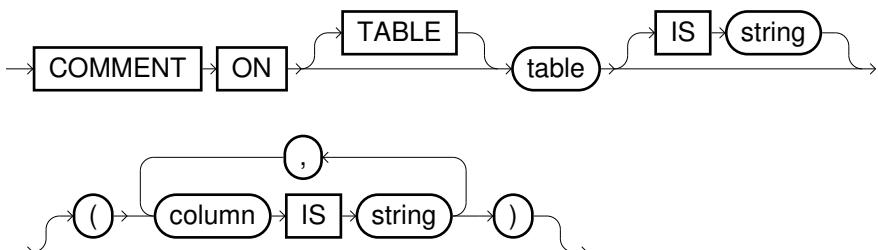
Prerequisite(s)

- Prerequisites for commenting an object:

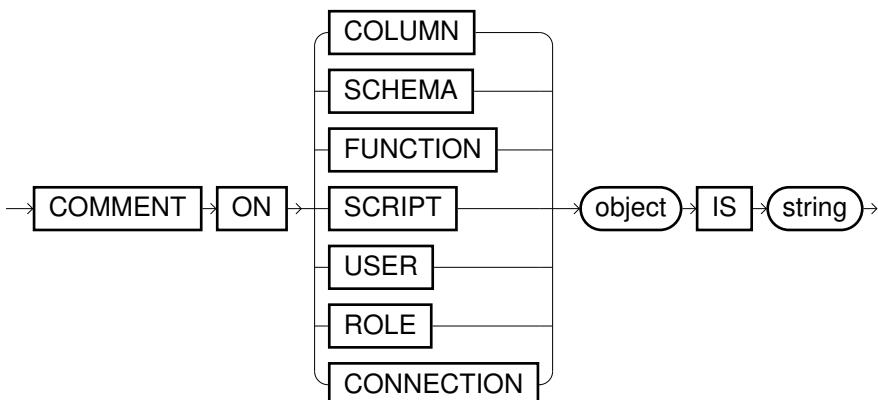
Object type	Prerequisite
Schema	Owner or role DBA
Table or column	Owner or role DBA, ALTER object privilege on the table or ALTER ANY TABLE.
Function	Owner or role DBA
Script	Owner or role DBA
User	Role DBA, system privilege ALTER USER or system privilege CREATE USER
Role	Role DBA or system privilege CREATE ROLE
Connection	Role DBA or system privilege CREATE CONNECTION

Syntax

comment_table ::=



comment_object ::=



Note(s)

- Via the corresponding metadata system tables (e.g. [EXA_ALL_OBJECTS](#), [EXA_ALL_TABLES](#), ...) and the command [DESC\[RIBE\]](#) (with FULL option) the comments can be displayed.
- Comments can be dropped by assigning NULL or the empty string.
- Comments can also be defined in the [CREATE TABLE](#) statement.
- View comments can only be specified in the [CREATE VIEW](#) statement.

Example(s)

```
COMMENT ON SCHEMA s1 IS 'My first schema';
COMMENT ON TABLE t1 IS 'My first table';
COMMENT ON t1 (id IS 'Identity column', zip IS 'Zip code');
COMMENT ON SCRIPT script1 IS 'My first script';
```

2.2.2. Manipulation of the database (DML)

The content of tables can be changed using the Data Manipulation Language (DML).

INSERT

Purpose

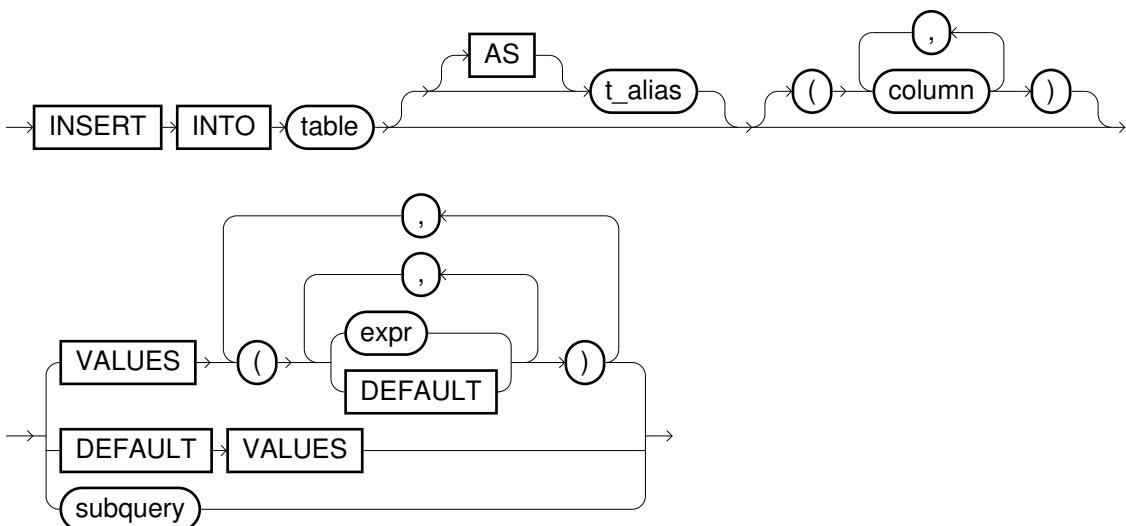
This statement makes it possible for the user to insert constant values as well the result of a subquery in a table.

Prerequisite(s)

- System privilege `INSERT ANY TABLE`, object privilege `INSERT` on the table or its schema, or the table belongs to the current user or one of his roles.
- If the result of a subquery is to be inserted, the user requires the appropriate `SELECT` rights for the objects referenced in the subquery.

Syntax

`insert ::=`



Note(s)

- The number of target columns of the target table must match the number of constants or the number of `SELECT` columns in a subquery. Otherwise, an exception occurs.
- If only a specific number of target columns (`<column_list>`) are specified for the target table, the entries of the remaining columns will be filled automatically. For identity columns a monotonically increasing number is generated and for columns with default values their respective default value is used. For all other columns the value `NULL` is inserted.
- If for `INSERT INTO t VALUES` the 'value' `DEFAULT` is specified for a column, then the behavior is the same as the implicit insert for unspecified columns (see above).
- `INSERT INTO t DEFAULT VALUES` has the same behavior as if you would specify the literal `DEFAULT` for each column.
- Details on default values can be found in [Section 2.3.5, “Default values”](#), on identity columns in [Section 2.3.6, “Identity columns”](#).
- Details about the syntax of subqueries can be found in the description of the `SELECT` statement in [Section 2.2.4, “Query language \(DQL\)”](#).

Example(s)

```
INSERT INTO t VALUES (1, 2.34, 'abc');
INSERT INTO t VALUES (2, 1.56, 'ghi'), (3, 5.92, 'pqr');
INSERT INTO t VALUES (4, DEFAULT, 'xyz');
INSERT INTO t (i,k) SELECT * FROM u;
INSERT INTO t (i) SELECT max(j) FROM u;
INSERT INTO t DEFAULT VALUES;
```

UPDATE

Purpose

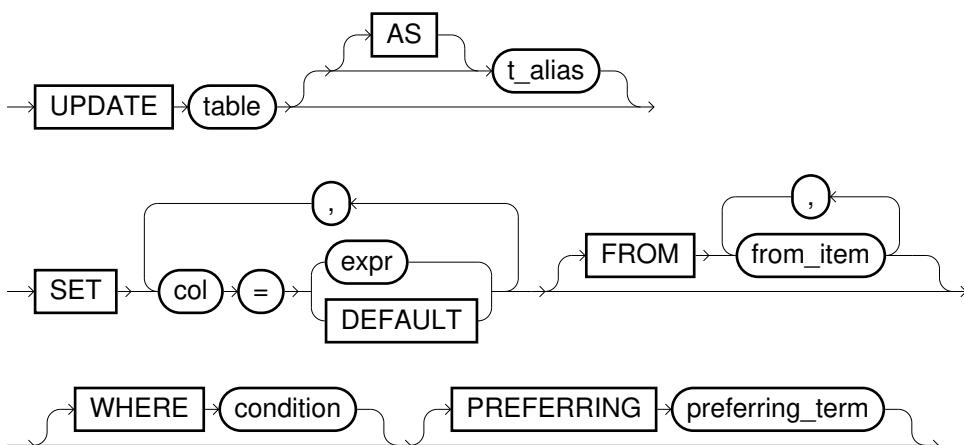
The UPDATE statement makes it possible to make targeted changes to the contents of a table. The restriction of the WHERE condition even makes it possible to change only one column entry of a single row.

Prerequisite(s)

- System privilege UPDATE ANY TABLE, object privilege UPDATE on the table or its schema, or the table belongs to the current user or one of his roles.
- Appropriate SELECT privileges on the schema objects referenced in the optional FROM clause.

Syntax

update ::=



Note(s)

- By using the FROM clause, you can define several tables which are joined through the WHERE clause. By that, you can specify complex update conditions which are similar to complex SELECT statements. Please note that you have to specify the updated table within the FROM clause.

Internally, this statement is transformed into a [MERGE](#) statement. Therefore, error messages can refer to the [MERGE](#) command. If one row is updated multiple times, the new value must be identical. Otherwise you will get the error message "Unable to get a stable set of rows in the source tables".

- If a column is set to the 'value' DEFAULT, then the rows in this column affected by the UPDATE are filled automatically. For identity columns a monotonically increasing number is generated and for columns with default values their respective default value is used. For all other columns the value NULL is used.

- Details on default values can be found in [Section 2.3.5, “Default values”](#), on identity columns in [Section 2.3.6, “Identity columns”](#).
- The `PREFERRING` clause defines an Skyline preference term. Details can be found in [Section 3.10, “Skyline”](#).

Example(s)

```
--Salary increase by 10 %
UPDATE staff SET salary=salary*1.1 WHERE name='SMITH';

--Euro conversion
UPDATE staff AS U SET U.salary=U.salary/1.95583, U.currency='EUR'
    WHERE U.currency='DM';

--Complex UPDATE using a join with another table
UPDATE staff AS U
SET U.salary=V.salary, U.currency=V.currency
FROM staff AS U, staff_updates AS V
WHERE U.name=V.name;
```

MERGE

Purpose

The `MERGE` statement makes it possible to import the contents of an update table into a target table. The rows of the update table determine which rows will be changed, deleted or inserted. Hence, the `MERGE` statement unites the three statements [UPDATE](#), [DELETE](#), and [INSERT](#).

For example, the update table can contain the data of new customers or customers to be dropped or the change information of already existing customers. The `MERGE` statement can now be used to insert the new customers into the target table, delete non-valid customers, and import the changes of existing customers.

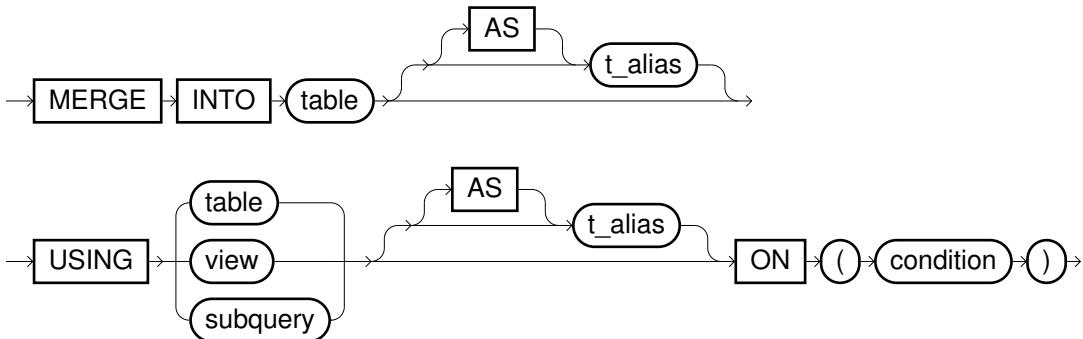
The `UPDATE`, `DELETE` and `INSERT` clauses are optional, which means that at any given time only one part of the actions described above is possible. Overall, the `MERGE` statement is a powerful tool for manipulating the database.

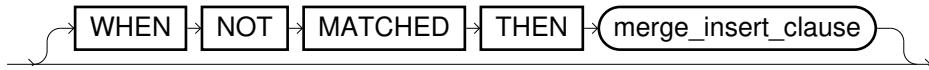
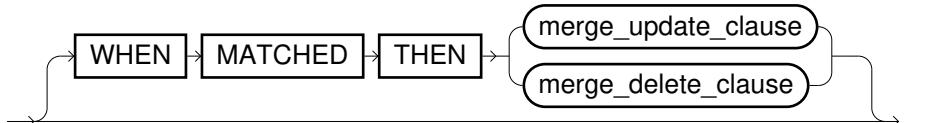
Prerequisite(s)

- Appropriate `INSERT`, `DELETE`, and `UPDATE` privileges on the target table.
- Appropriate `SELECT` privileges on the update table.

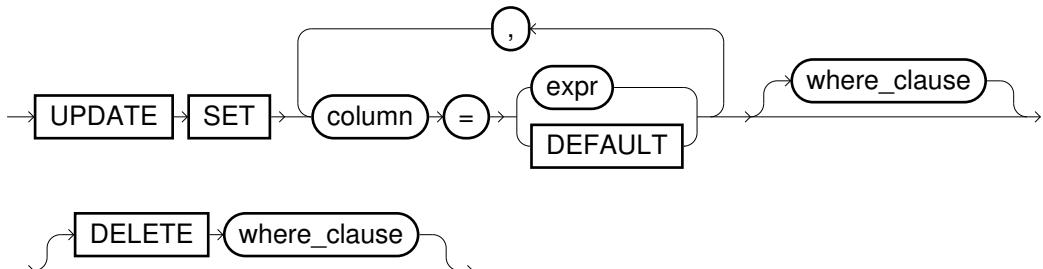
Syntax

`merge ::=`

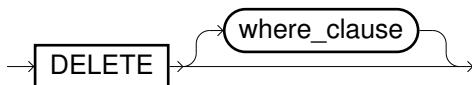




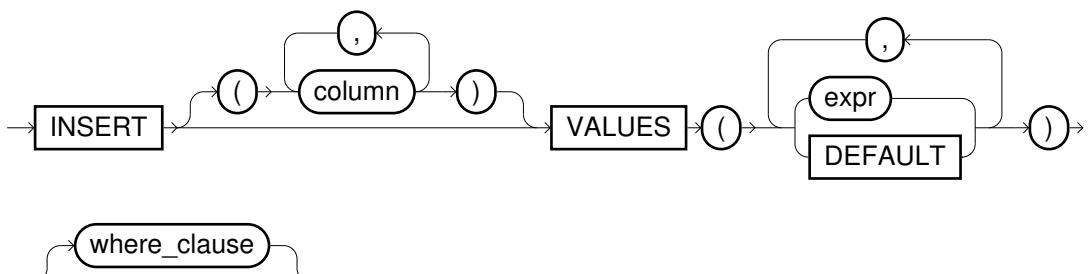
merge_update_clause::=



merge_delete_clause::=



merge_insert_clause::=



Note(s)

- The **ON condition** describes the correlation between the two tables (similar to a join). The MATCHED clause is used for matching row pairs, the NOT MATCHED clause is used for those that do not match. In the ON condition only equivalence conditions ($=$) are permitted.
- **UPDATE clause:** the optional WHERE condition specifies the circumstances under which the UPDATE is conducted, whereby it is permissible for both the target table and the change table to be referenced for this. With the aid of the optional DELETE condition it is possible to delete rows in the target table. Only the rows that have been changed are taken into account and used to check the values *after* the UPDATE.
- **DELETE clause:** the optional WHERE condition specifies the circumstances under which the DELETE is conducted.
- **INSERT clause:** the optional WHERE condition specifies the circumstances under which the INSERT is conducted. In this respect, it is only permissible to reference the columns of the change table.
- The change table can be a physical table, a view or a subquery.
- The UPDATE or DELETE and INSERT clauses are optional with the restriction that at least one must be specified. The order of the clauses can be exchanged.
- Default values and identity columns are treated by the INSERT and UPDATE clauses in exactly the same way as by the INSERT and UPDATE statements (see there), with the only exception that INSERT DEFAULT VALUES is not allowed.

- If there are several entries in the change table that could apply to an UPDATE of a single row in the target table, this leads to the error message "Unable to get a stable set of rows in the source tables" if the original value of the target table would be changed by the UPDATE candidates.
- If there are several entries in the change table that could apply to a DELETE of a single row in the target table, this leads to the error message "Unable to get a stable set of rows in the source tables".

Example(s)

```
/* Sample tables
staff:                                changes:          deletes:
 name   | salary | lastChange      name   | salary
-----+-----+-----+-----+-----+-----+
 meier | 30000 | 2006-01-01    schmidt | 43000
 schmidt | 40000 | 2006-05-01  hofmann | 35000
 mueller | 50000 | 2005-08-01  meier   | 29000
 */

-- Merging the table updates
MERGE INTO staff T
  USING changes U
  ON (T.name = U.name)
  WHEN MATCHED THEN UPDATE SET T.salary = U.salary,
                               T.lastChange = CURRENT_DATE
                               WHERE T.salary < U.salary
  WHEN NOT MATCHED THEN INSERT VALUES (U.name,U.salary,CURRENT_DATE);

SELECT * FROM staff;

NAME          SALARY        LASTCHANGE
-----+-----+-----+-----+
meier          30000 2006-01-01
schmidt        43000 2010-10-06
mueller        50000 2005-08-01
hofmann        35000 2010-10-06

-- Merging the table deletes
MERGE INTO staff T
  USING deletes U
  ON (T.name = U.name)
  WHEN MATCHED THEN DELETE;

SELECT * FROM staff;

NAME          SALARY        LASTCHANGE
-----+-----+-----+-----+
hofmann        35000 2010-10-06
schmidt        43000 2010-10-06
mueller        50000 2005-08-01
```

DELETE

Purpose

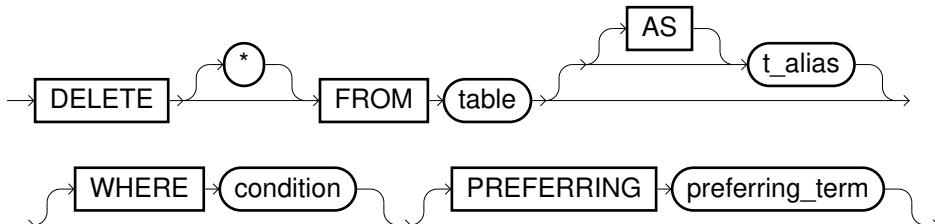
The DELETE statement makes it possible to delete rows in a table.

Prerequisite(s)

- System privilege `DELETE ANY TABLE`, object privilege `DELETE` on the table or its schema, or the table belongs to the current user or one of his roles.

Syntax

`delete ::=`



Note(s)

- Internally, rows are not immediately deleted, but marked as deleted. When a certain threshold is reached (default is 25% of the rows), these rows are finally dropped. Hence `DELETE` statements can have varying execution times. The current percentage of marked rows can be found in system tables `EXA_*_TABLES` (e.g. `EXA_ALL_TABLES`) in the column `DELETE_PERCENTAGE`.
- The `PREFERRING` clause defines an Skyline preference term. Details can be found in [Section 3.10, “Skyline”](#).

Example(s)

```
DELETE FROM staff WHERE name='SMITH';
DELETE FROM staff;
```

TRUNCATE

Purpose

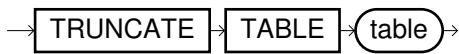
The `TRUNCATE` statement makes it possible to completely delete the contents of a table.

Prerequisite(s)

- System privilege `DELETE ANY TABLE`, object privilege `DELETE` on the table or its schema, or the table belongs to the current user or one of his roles.

Syntax

`truncate ::=`



Example(s)

```
TRUNCATE TABLE staff;
```

IMPORT

Purpose

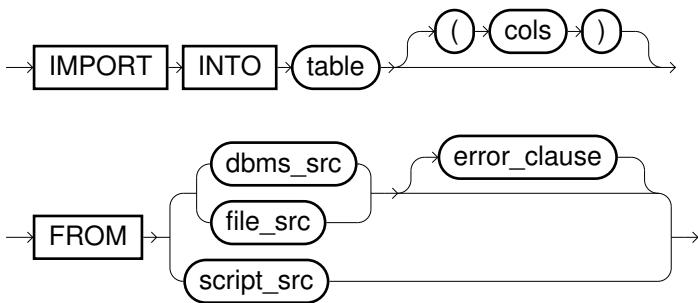
Via the IMPORT command you can transfer data from external data sources into a table.

Prerequisite(s)

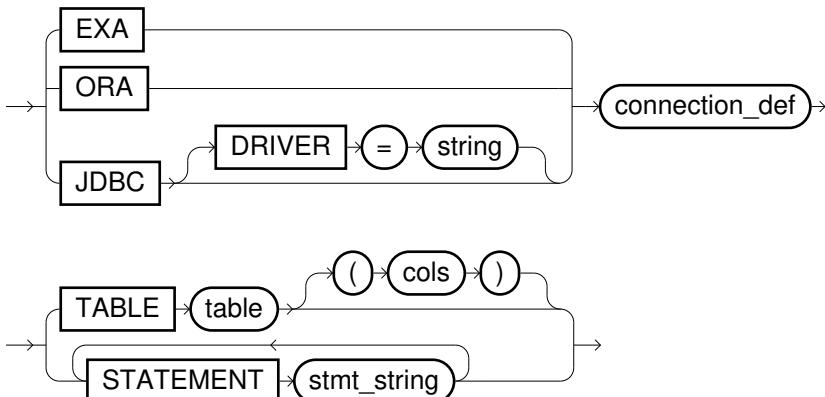
- In the source system: corresponding privileges to read the table contents or the files
- In Exasol: corresponding privileges to insert rows into the table (see [INSERT](#))
- When using a connection (see also [CREATE CONNECTION](#) in [Section 2.2.3, “Access control using SQL \(DCL\)”](#)) you need either the system privilege USE ANY CONNECTION or the connection has to be granted via the [GRANT](#) statement to the user or to one of its roles
- When using an error table you need the appropriate rights for writing or inserting data

Syntax

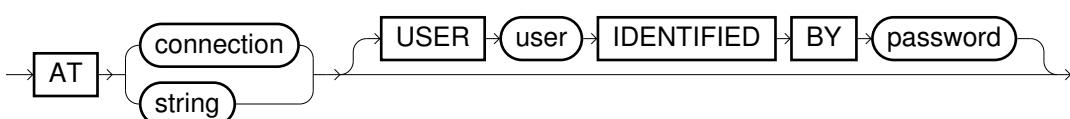
import ::=



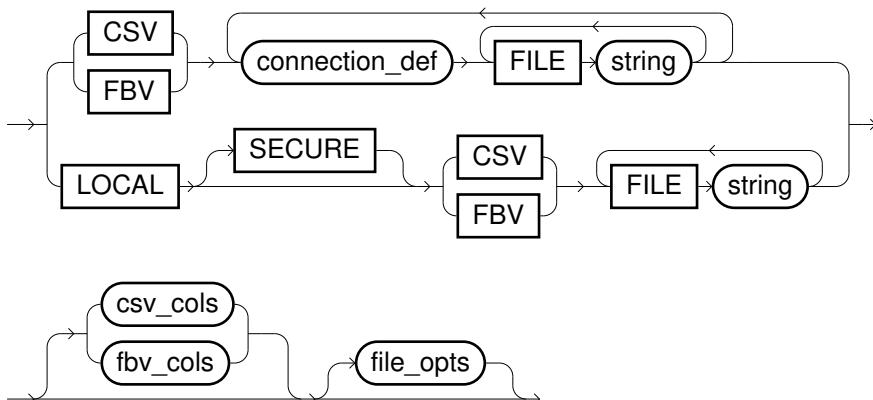
dbms_src ::=



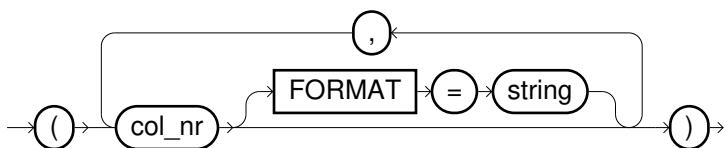
connection_def ::=



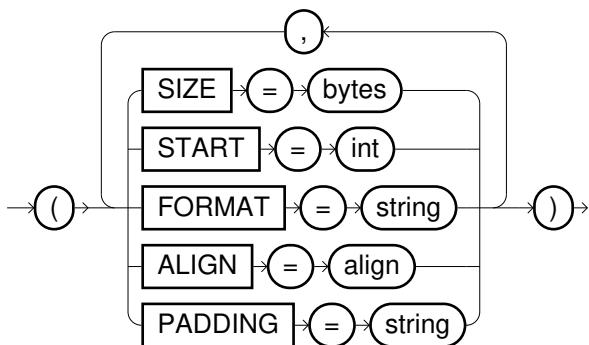
file_src ::=



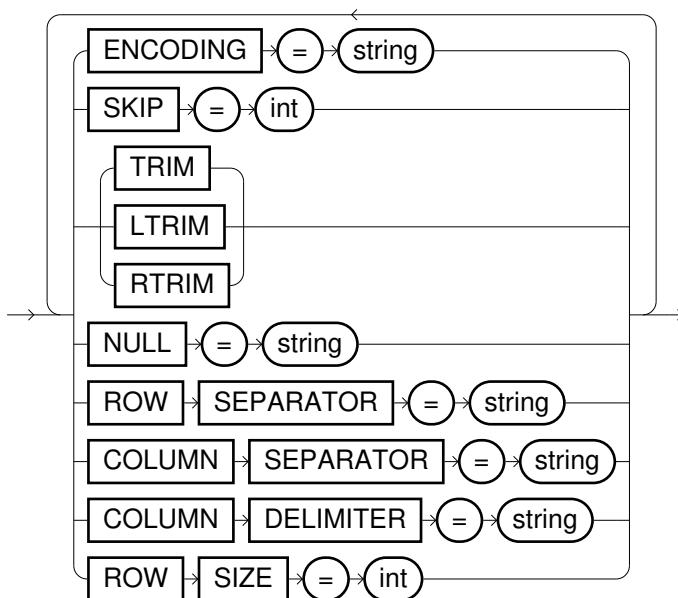
`csv_cols:=`



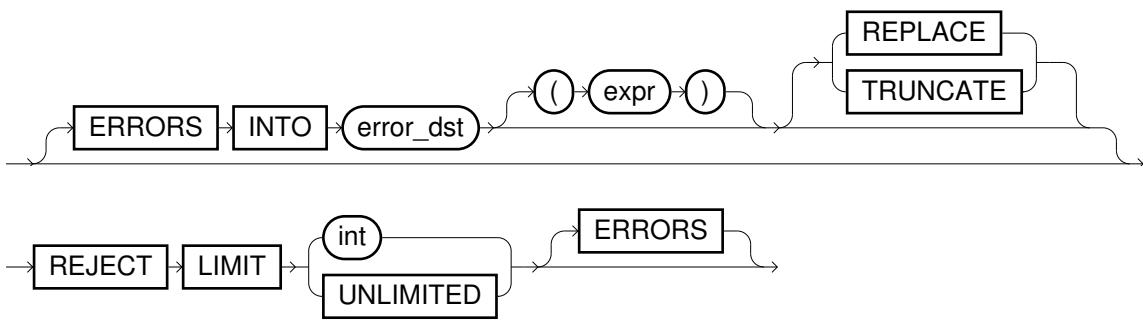
`fbv_cols:=`



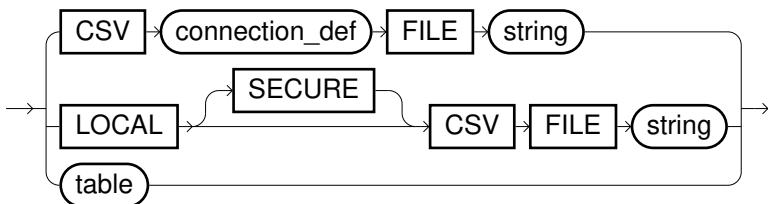
`file_opts:=`



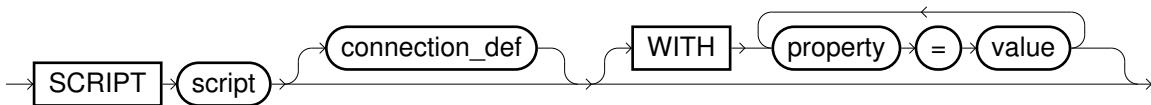
error_clause:=



error_dst:=



script_src:=



Note(s)

- Additional information about ETL processes can be found in [Section 3.4, “ETL Processes”](#)
- The current progress of the data transfer can be seen within a second connection via the system table `EXA_USER_SESSIONS` (column ACTIVITY)
- Import statements can also be transparently used within SELECT queries. Further details can be found in [Section 2.2.4, “Query language \(DQL\)”](#).
- Please note that in case of an IMPORT from JDBC or CSV sources, decimals are truncated if the target data type has less precision than the source data type.
- Overview of the different elements and their meaning:

Element	Meaning
dbms_src	<p>Defines the database source whose connection data is specified in the <code>connection_def</code> (see below). You can choose among an Exasol connection (EXA), a native connection to an Oracle database (ORA) or a JDBC connection to any database (JDBC).</p> <p>Some JDBC drivers are already delivered as default (visible in EXAoperation) and can be addressed within the connection string (e.g. <code>jdbc:mysql</code>, <code>jdbc:postgres</code>). You can additionally configure JDBC drivers in EXAoperation and choose them via the DRIVER option if its prefix is ambiguous.</p> <p> Only the pre-installed JDBC drivers (marked gray in EXAoperation) are tested and officially supported. But our support will try to help you in case of problems with other drivers.</p> <p>The source data can either be a database table (as identifier like e.g. <code>MY_SCHEMA.MY_TABLE</code>) or a database statement (as string like e.g. '<code>SELECT</code></p>

Element	Meaning
	<p>"TEST" FROM DUAL'). In the second case this expression is executed on the database, e.g. a SQL query or a procedure call.</p> <p> Please note that table names are treated similar to Exasol tables. Therefore you have to quote case-sensitive table names.</p> <p>Please note the following to achieve an optimal parallelization:</p> <ul style="list-style-type: none"> Importing from Exasol databases is always parallelized. Please note that for Exasol, loading tables directly is significantly faster than using database statements. If you import data from Oracle sources, partitioned tables will be loaded in parallel. Specifying multiple statements is only possible for JDBC and Oracle sources.
<code>file_src</code>	<p>Specifies the data file source.</p> <ol style="list-style-type: none"> Remote file(s) <ul style="list-style-type: none"> FTP, FTPS, SFTP, HTTP and HTTPS servers are supported whose connection data is defined via the <code>connection_def</code> (see below). <p>Notes:</p> <ul style="list-style-type: none"> Certificates are not verified for encrypted connections. If you specify a folder instead of a file name, then the list of contained files will be imported if the server supports that. In case of URLs starting with '<code>ftps://</code>', the implicit encryption is used. In case of URLs starting with '<code>ftp://</code>', Exasol encrypts user name and password (explicit encryption) if this is supported by the server. If the server demands encryption, then the whole data transfer is done encrypted. For HTTP and HTTPS servers only basic authentication is supported. For HTTP and HTTPS connections, http query parameters can be specified by appending them to the file name (e.g. <code>FILE 'file.csv?op=OPEN&user.name=user'</code>). Local file(s) <ul style="list-style-type: none"> You can also import local files from your client system. When specifying the <code>SECURE</code> option, the data is transferred encrypted, but also with slower performance. <ul style="list-style-type: none">  This functionality is only supported for EXAplus and the JDBC driver and can neither be used in prepared statements nor within database scripts. If you want to process a local file via an explicit program, you can use the tool EXAjload which is delivered within the JDBC driver package. Execute this program without parameters to get information about the usage.  For importing local files, the JDBC driver opens an internal connection to the cluster and provides a HTTP or HTTPS (SECURE-Option) server. But this is all transparent for the user. <p>The source files can either be CSV or FBV files and shall comply to the format specifications in The CSV Data format and The Fixblock Data format (FBV). File names may only consist of ASCII characters. A BOM is not supported.</p>

Element	Meaning				
	<p>Compressed files are recognized by their file extension. Supported extensions are <code>.zip</code>, <code>.gz</code> (gzip) and <code>.bz2</code> (bzip2).</p> <p>When <code>System.in</code> is specified as filename, data is read from the standard input stream (<code>System.in</code>).</p>				
<code>script_src</code>	<p>Specifies the UDF script to be used for a user-defined import. Optionally, you can define a connection or properties which will be forwarded to the script. The specified script will generate an SQL statement internally which does the actual import using <code>INSERT INTO SELECT</code>. The script has to implement a special callback function which receives the import specification (e.g. parameters and connection information) and returns an SQL statement. For details and examples, we refer to Section 3.4.4, “User-defined IMPORT using UDFs”.</p> <p>connection_def</p> <p>Optional connection definition for being able to encapsulate connection information such as password. See also the separate section in this table for the exact syntax.</p> <p>WITH parameter=value ...</p> <p>Optional parameters to be passed to the script. Each script can define the mandatory and optional parameters it supports. Parameters are simple key-value pairs, with value being a string, e.g.:</p> <pre>... WITH PARAM_1='val1' PARAM_2 = 'val2';</pre>				
<code>connection_def</code>	<p>Defines the connection to the external database or file server. This can be specified within a connection string (e.g. <code>'ftp://192.168.1.1/'</code>) and the corresponding login information.</p> <p>For regular ETL jobs you can also take use of connections where the connection data like user name and password can easily be encapsulated. For details and examples please refer to CREATE CONNECTION in Section 2.2.3, “Access control using SQL (DCL)”.</p> <p>The declaration of user and password within the IMPORT command are optional. If they are omitted, the data of the connection string or the connection object are used.</p> <p>For JDBC connections, it is possible to use Kerberos authentication by specifying specific data in the <code>IDENTIFIED BY</code> field. This data consists of a key which indicates that Kerberos authentication should be used (<code>ExaAuthType=Kerberos</code>), a base64 encoded configuration file and a base64 encoded keytab file containing the credentials for the principal. The syntax looks like the following: <code>IMPORT INTO table1 FROM JDBC AT '<JDBC_URL>' USER '<kerberos_principal>' IDENTIFIED BY 'ExaAuthType=Kerberos ;<base64_krb_conf> ;<base64_keytab>'</code> TABLE <code>table2</code>; Further details and examples can be found in our solution center: https://wwwexasolcom/portal/display/SOL-512</p>				
<code>csv_cols</code>	<p>Defines which and how the columns of the CSV file are interpreted. Please also refer to The CSV Data format.</p> <table> <tr> <td><code>col_nr</code></td> <td>Defines the column number (starting from 1). Alternatively, you can define a certain column range via <code>..</code> (e.g. <code>5..8</code> for columns 5,6,7,8). Please note that column numbers have to be in ascending order!</td> </tr> <tr> <td><code>FORMAT</code></td> <td>Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models”.</td> </tr> </table>	<code>col_nr</code>	Defines the column number (starting from 1). Alternatively, you can define a certain column range via <code>..</code> (e.g. <code>5..8</code> for columns 5,6,7,8). Please note that column numbers have to be in ascending order!	<code>FORMAT</code>	Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models” .
<code>col_nr</code>	Defines the column number (starting from 1). Alternatively, you can define a certain column range via <code>..</code> (e.g. <code>5..8</code> for columns 5,6,7,8). Please note that column numbers have to be in ascending order!				
<code>FORMAT</code>	Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models” .				

Element	Meaning										
	<p>In the following example the first 4 columns of the CSV file is loaded, the last column with the specified date format:</p> <pre>(1 .. 3 , 4 FORMAT= ' DD-MM-YYYY')</pre>										
fbv_cols	<p>Defines which and how the columns of the FBV file are interpreted. Please also refer to The Fixblock Data format (FBV).</p> <p>The following elements can be specified:</p> <table> <tr> <td>SIZE</td><td>Defines the number of bytes of the column and must always be specified.</td></tr> <tr> <td>START</td><td>Start byte of the column (starting with 0). Note that the START values have to be in ascending order!</td></tr> <tr> <td>FORMAT</td><td>Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models”.</td></tr> <tr> <td>ALIGN</td><td>Alignment of the column (LEFT or RIGHT), default is LEFT.</td></tr> <tr> <td>PAD-DING</td><td>Padding characters for columns. In the default case, Space is used. You can specify any ASCII character, either in plain text (e.g.: '+'), as hexadecimal value (e.g: '0x09') or as abbreviation (one of 'NUL','TAB','LF','CR','ESC').</td></tr> </table> <p>In the following example 4 columns of a FBV file are imported. The first column is right-aligned and padded with x characters. After the first 12 bytes a gap exists and the fourth column has the specified date format:</p> <pre>(SIZE=8 PADDING= 'x' ALIGN=RIGHT, SIZE=4, START=17 SIZE=8, SIZE=32 FORMAT= ' DD-MM-YYYY')</pre>	SIZE	Defines the number of bytes of the column and must always be specified.	START	Start byte of the column (starting with 0). Note that the START values have to be in ascending order!	FORMAT	Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models” .	ALIGN	Alignment of the column (LEFT or RIGHT), default is LEFT.	PAD-DING	Padding characters for columns. In the default case, Space is used. You can specify any ASCII character, either in plain text (e.g.: '+'), as hexadecimal value (e.g: '0x09') or as abbreviation (one of 'NUL','TAB','LF','CR','ESC').
SIZE	Defines the number of bytes of the column and must always be specified.										
START	Start byte of the column (starting with 0). Note that the START values have to be in ascending order!										
FORMAT	Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models” .										
ALIGN	Alignment of the column (LEFT or RIGHT), default is LEFT.										
PAD-DING	Padding characters for columns. In the default case, Space is used. You can specify any ASCII character, either in plain text (e.g.: '+'), as hexadecimal value (e.g: '0x09') or as abbreviation (one of 'NUL','TAB','LF','CR','ESC').										
file_opts	<table> <tr> <td>ENCODING</td><td>Encoding of the CSV or FBV file (default is UTF8). All supported encodings can be found in Appendix D, Supported Encodings for ETL processes and EXAplus.</td></tr> <tr> <td>ROW SEPARATOR</td><td>Line break character: <ul style="list-style-type: none"> 'LF' (Default): corresponds to the ASCII character 0x0a 'CR': corresponds to the ASCII character 0x0d 'CRLF': corresponds to the ASCII characters 0x0d and 0x0a 'NONE': no line break (only allowed for FBV files) </td></tr> <tr> <td>NULL</td><td>Representation of NULL values. If you didn't specify that option, NULL values are represented as the empty string.</td></tr> <tr> <td>SKIP</td><td>Number of rows which shall be omitted. This can be useful if you have included header information within the data files. You have to mention that SKIP corresponds to the number of line breaks (ROW SEPARATOR), also if they occur inside the data.</td></tr> <tr> <td>TRIM, LTRIM, RTRIM</td><td>Defines whether spaces are deleted at the border of CSV columns (LTRIM: from the left, RTRIM: from the right,</td></tr> </table>	ENCODING	Encoding of the CSV or FBV file (default is UTF8). All supported encodings can be found in Appendix D, Supported Encodings for ETL processes and EXAplus .	ROW SEPARATOR	Line break character: <ul style="list-style-type: none"> 'LF' (Default): corresponds to the ASCII character 0x0a 'CR': corresponds to the ASCII character 0x0d 'CRLF': corresponds to the ASCII characters 0x0d and 0x0a 'NONE': no line break (only allowed for FBV files) 	NULL	Representation of NULL values. If you didn't specify that option, NULL values are represented as the empty string.	SKIP	Number of rows which shall be omitted. This can be useful if you have included header information within the data files. You have to mention that SKIP corresponds to the number of line breaks (ROW SEPARATOR), also if they occur inside the data.	TRIM, LTRIM, RTRIM	Defines whether spaces are deleted at the border of CSV columns (LTRIM: from the left, RTRIM: from the right,
ENCODING	Encoding of the CSV or FBV file (default is UTF8). All supported encodings can be found in Appendix D, Supported Encodings for ETL processes and EXAplus .										
ROW SEPARATOR	Line break character: <ul style="list-style-type: none"> 'LF' (Default): corresponds to the ASCII character 0x0a 'CR': corresponds to the ASCII character 0x0d 'CRLF': corresponds to the ASCII characters 0x0d and 0x0a 'NONE': no line break (only allowed for FBV files) 										
NULL	Representation of NULL values. If you didn't specify that option, NULL values are represented as the empty string.										
SKIP	Number of rows which shall be omitted. This can be useful if you have included header information within the data files. You have to mention that SKIP corresponds to the number of line breaks (ROW SEPARATOR), also if they occur inside the data.										
TRIM, LTRIM, RTRIM	Defines whether spaces are deleted at the border of CSV columns (LTRIM: from the left, RTRIM: from the right,										

Element	Meaning				
	TRIM: from both sides). In default case, no spaces are trimmed.				
COLUMN SEPARATOR	Defines the field separator for CSV files. In the default case, the comma (,) is used. You can specify any string, either as plain text (e.g.: ','), as a hexadecimal value (e.g.: '0x09') or as an abbreviation (one of 'NUL', 'TAB', 'LF', 'CR', 'ESC'). A plain text value is limited to 10 characters, which will be automatically converted to the file's specified ENCODING (see above). A hexadecimal value is limited to 10 bytes (not characters) and will not be converted.				
COLUMN DELIMITER	Defines the field delimiter for CSV files. In the default case, the double quote ("") is used. You can specify any string, either as plain text (e.g.: ""), as a hexadecimal value (e.g.: '0x09') or as an abbreviation (one of 'NUL', 'TAB', 'LF', 'CR', 'ESC'). A plain text value is limited to 10 characters, which will be automatically converted to the file's specified ENCODING (see above). A hexadecimal value is limited to 10 bytes (not characters) and will not be converted. If you don't want to use any field delimiter, you can define the empty string ("").				
ROW SIZE	Only for FBV files. If the last column of the FBV file is not used, this value must be specified to recognize the end of a row. Otherwise the end is implicitly calculated by the last column which was defined in the IMPORT command (e.g. in case of (SIZE=4 START=5) it is assumed that one column is read with 4 bytes and that the row consists of overall 9 bytes).				
error_clause	<p>This clause defines how many invalid rows of the source are allowed. E.g. in case of REJECT LIMIT 5 the statement would work fine if less or equal than five invalid rows occur, and would throw an exception after the sixth error.</p> <p>Additionally you can write the faulty rows into a file or a local table within Exasol to process or analyze them afterwards:</p> <table> <tr> <td>Table</td> <td>For every faulty row, the following columns are created: row number, error message, (expression), truncated flag and the actual data. The truncated flag indicates whether the data was truncated to the maximal string length.</td> </tr> <tr> <td>CSV file</td> <td>For every faulty row, a comment row is created with row number, error message and (expression), followed by the actual data row.</td> </tr> </table> <p>The (optional) expression can be specified for identification reasons in case you use the same error table or file multiple times. You could e.g. use CURRENT_TIMESTAMP for that.</p>	Table	For every faulty row, the following columns are created: row number, error message, (expression), truncated flag and the actual data. The truncated flag indicates whether the data was truncated to the maximal string length.	CSV file	For every faulty row, a comment row is created with row number, error message and (expression), followed by the actual data row.
Table	For every faulty row, the following columns are created: row number, error message, (expression), truncated flag and the actual data. The truncated flag indicates whether the data was truncated to the maximal string length.				
CSV file	For every faulty row, a comment row is created with row number, error message and (expression), followed by the actual data row.				

Example(s)

```
IMPORT INTO table_1 FROM CSV
  AT 'http://192.168.1.1:8080/' USER 'agent_007' IDENTIFIED BY 'secret'
  FILE 'tabl_part1.csv' FILE 'tabl_part2.csv'
  COLUMN SEPARATOR = ';'
```

```
SKIP = 5;

CREATE CONNECTION my_fileserver
    TO 'ftp://192.168.1.2/' USER 'agent_007' IDENTIFIED BY 'secret';

IMPORT INTO table_2 FROM FBV
    AT my_fileserver
    FILE 'tab2_part1.fbv'
    ( SIZE=8 PADDING='+' ALIGN=RIGHT,
      SIZE=4,
      SIZE=8,
      SIZE=32 FORMAT='DD-MM-YYYY' );

CREATE CONNECTION my_oracle
    TO '(DESCRIPTION =
        (ADDRESS_LIST = (ADDRESS =
            (PROTOCOL = TCP)
            (HOST = 192.168.0.25)(PORT = 1521)
        )
    )
    (CONNECT_DATA = (SERVICE_NAME = orautf8)))
)';

IMPORT INTO table_3 (col1, col2, col4) FROM ORA
    AT my_oracle
    USER 'agent_008' IDENTIFIED BY 'secret'
    STATEMENT ' SELECT * FROM orders WHERE order_state=''OK''''
    ERRORS INTO error_table (CURRENT_TIMESTAMP) REJECT LIMIT 10;

IMPORT INTO table_4 FROM JDBC
    AT 'jdbc:exa:192.168.6.11..14:8563'
    USER 'agent_008' IDENTIFIED BY 'secret'
    STATEMENT ' SELECT * FROM orders WHERE order_state=''OK'''';

IMPORT INTO table_5 FROM CSV
    AT 'http://HadoopNode:50070/webhdfs/v1/tmp'
    FILE 'file.csv?op=OPEN&user.name=user';

IMPORT INTO table_6 FROM CSV
    AT 'https://testbucket.s3.amazonaws.com'
    USER '<AccessKeyID>' IDENTIFIED BY '<SecretAccessKey>'
    FILE 'file.csv';

IMPORT INTO table_7 FROM EXA
    AT my_exasol
    TABLE MY_SCHEMA.MY_TABLE;

IMPORT INTO table_8 FROM SCRIPT etl.import_hcat_table
WITH HCAT_DB      = 'default'
      HCAT_TABLE   = 'my_hcat_table'
      HCAT_ADDRESS = 'hcatalog-server:50111'
      HDFS_USER    = 'hdfs';IMPORT INTO table_9
      FROM LOCAL CSV FILE '~/my_table.csv'
      COLUMN SEPARATOR = ';' SKIP = 5;
```

EXPORT

Purpose

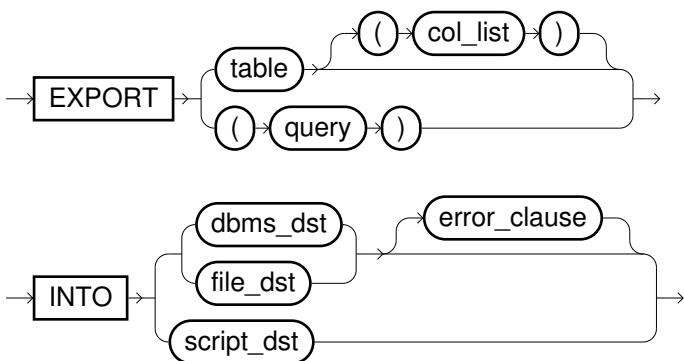
Via the EXPORT command you can transfer data from Exasol into an external files or database systems.

Prerequisite(s)

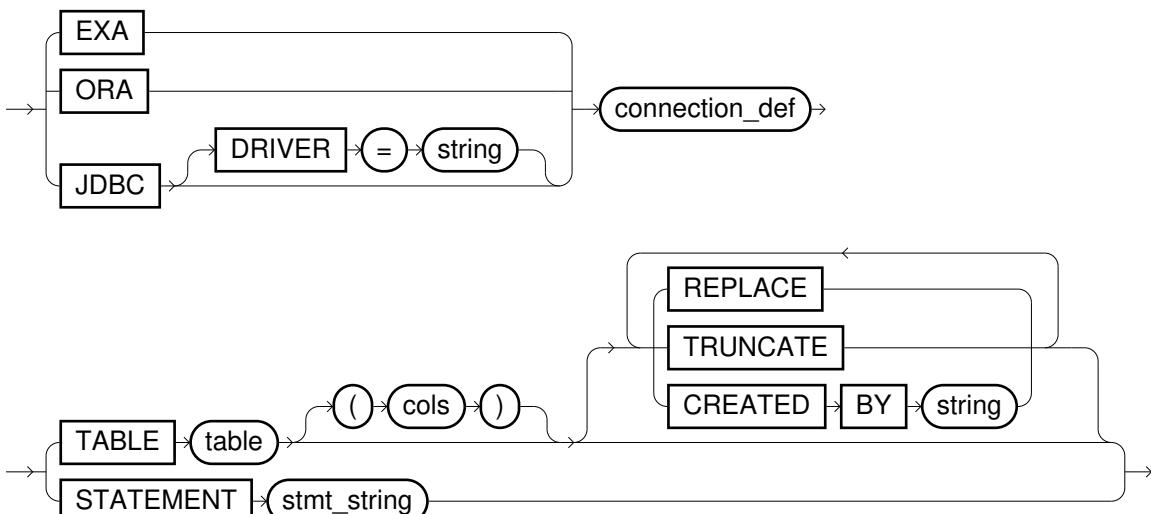
- In the target system: corresponding privileges to insert rows or writing files. If you specify the corresponding options, you need rights to replace or truncate the target.
- In Exasol: corresponding privileges to read the table contents.
- When using a connection (see also [CREATE CONNECTION](#) in [Section 2.2.3, “Access control using SQL \(DCL\)“](#)) you need either the system privilege USE ANY CONNECTION or the connection has to be granted via the [GRANT](#) statement to the user or to one of its roles

Syntax

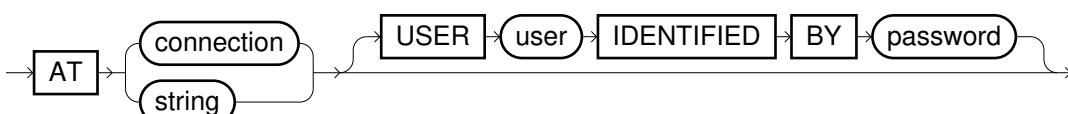
export::=



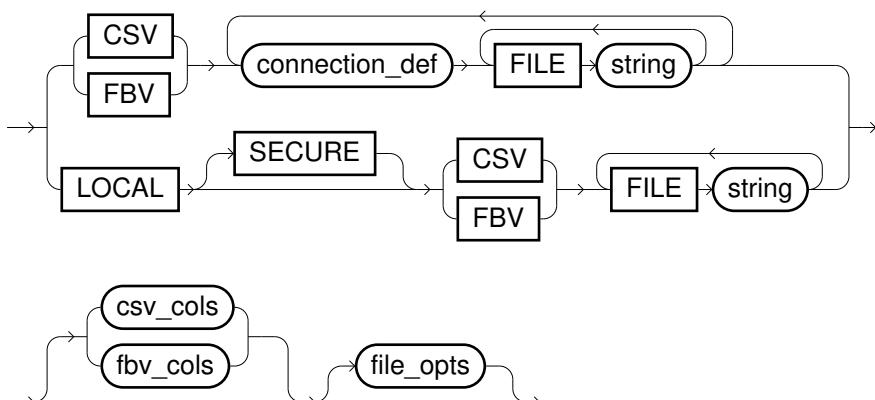
dbms_dst::=



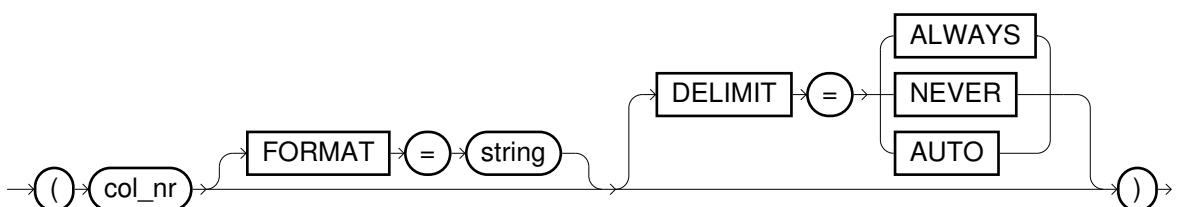
connection_def::=



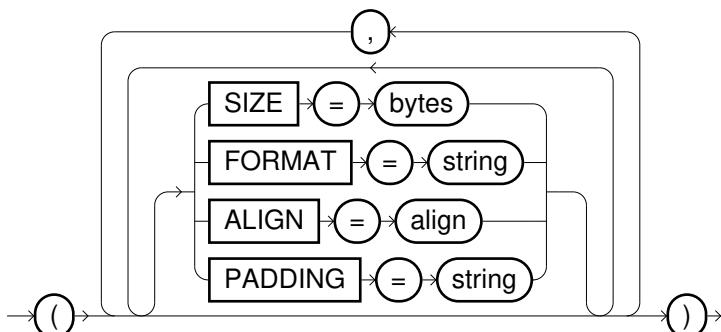
file_dst:=



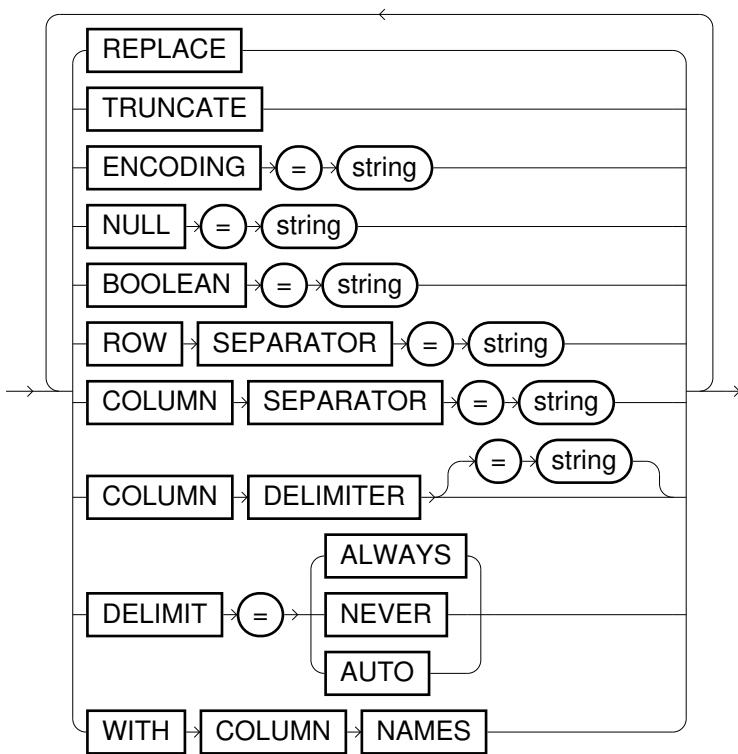
`csv_cols`:



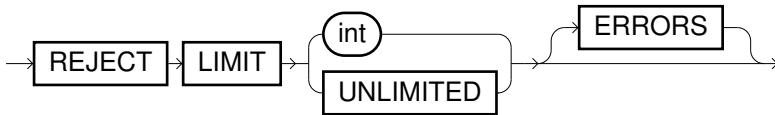
`fbv_cols`:



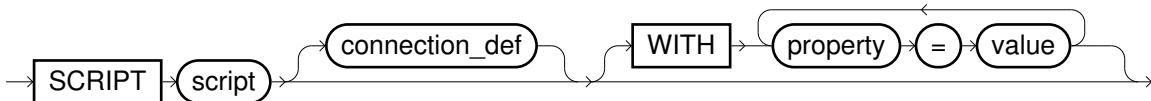
`file_opts`:



error_clause:=



script_dst:=



Note(s)

- Additional information about ETL processes can be found in [Section 3.4, “ETL Processes”](#)
- If no other option is specified (see below), the data is appended to the target
- The current progress of the data transfer can be seen within a second connection via the system table `EXA_USER_SESSIONS` (column ACTIVITY)
- Only statements or views with ORDER BY clause on the top level are exported in sorted order (only in case of files).
- Overview of the different elements and their meaning:

Element	Meaning
data_src	The source data can either be a table (as identifier like e.g. MY_SCHEMA . MY_TABLE) or a query (as string like e.g. 'SELECT "TEST" FROM DUAL'). For tables you can also specify the columns to be used.
dbms_dst	Defines the database destination whose connection data is specified in the connection_def (see below). You can choose among an Exasol connection (EXA), a native connection to an Oracle database (ORA) or a JDBC connection to any database (JDBC).

Element	Meaning
	<p>Some JDBC drivers are already delivered as default (visible in EXAoperation) and can be addressed within the connection string (e.g. jdbc:mysql, jdbc:postgres). You can additionally configure JDBC drivers in EXAoperation and choose them via the DRIVER option if its prefix is ambiguous.</p> <p> Only the pre-installed JDBC drivers (marked gray in EXAoperation) are tested and officially supported. But our support will try to help you in case of problems with other drivers.</p> <p>For the target you can define either a table or a prepared statement (e.g. an INSERT statement or a procedure call). In the latter case the data is passed as input data to the prepared statement. Please note that you have to use schema-qualified table names.</p> <p> Please note that table names are treated similar to Exasol tables. Therefore you have to quote case-sensitive table names.</p> <p>Further options for the target:</p> <ul style="list-style-type: none"> REPLACE Deletes the target table before the export is started TRUNCATE Deletes all rows of the target table before the export is started. TRUNCATE cannot be combined with REPLACE nor CREATED BY! CREATED BY Defines a creation string which is used to create the table before the export is started
<code>file_dst</code>	<p>Specifies the data file target.</p> <ol style="list-style-type: none"> 1. Remote file <p>FTP, FTPS, SFTP, HTTP and HTTPS servers are supported whose connection data is defined via the <code>connection_def</code> (see below).</p> <p>Notes:</p> <ul style="list-style-type: none"> • Certificates are not verified for encrypted connections. • In case of URLs starting with '<code>f tps://</code>', the implicit encryption is used. • In case of URLs starting with '<code>f tp://</code>', Exasol encrypts user name and password (explicit encryption) if this is supported by the server. If the server demands encryption, then the whole data transfer is done encrypted. • For HTTP and HTTPS servers only basic authentication is supported. • For HTTP and HTTPS connections, http query parameters can be specified by appending them to the file name (e.g. <code>FILE 'file.csv?op=CREATE&user.name=user'</code>). <ol style="list-style-type: none"> 2. Local file <p>You can also export into local files on your client system. When specifying the SECURE option, the data is transferred encrypted, but also with slower performance.</p> <p> This functionality is only supported for EXAplus and the JDBC driver and can neither be used in prepared statements nor within database scripts. If you want to process a local file via an explicit program, you can use the tool EXAjload which is delivered within the JDBC driver package. Execute this program without parameters to get information about the usage.</p>

Element	Meaning
	<p> For exporting local files, the JDBC driver opens an internal connection to the cluster and provides a HTTP or HTTPS (SECURE-Option) server. But this is all transparent for the user.</p> <p>The target file can either be CSV or FBV files and shall comply to the format specifications in The CSV Data format and The Fixblock Data format (FBV). File names may only consist of ASCII characters. A BOM is not supported.</p> <p>Compressed files are recognized by their file extension. Supported extensions are <code>.zip</code>, <code>.gz</code> (gzip) and <code>.bz2</code> (bzip2).</p> <p>When specifying multiple files, the actual data distribution depends on several factors. It is also possible that some file are completely empty.</p>
script_dst	<p>Specifies the UDF script to be used for a user-defined export. Optionally, you can define a connection or properties which will be forwarded to the script. The specified script will generate a SELECT statement internally that will be executed to do the actual export. The script has to implement a special callback function that receives the export specification (e.g. parameters and connection information) and returns a SELECT statement. For details and examples, refer to Section 3.4.5, “User-defined EXPORT using UDFs”.</p> <p>connection_def</p> <p>Optional connection definition for being able to encapsulate connection information such as password. See also the separate section in this table for the exact syntax.</p> <p>WITH parameter=value ...</p> <p>Optional parameters to be passed to the script. Each script can define the mandatory and optional parameters it supports. Parameters are simple key-value pairs, with value being a string, e.g.:</p> <pre data-bbox="446 1304 1343 1338">... WITH PARAM_1='val1' PARAM_2 = 'val2';</pre>
connection_def	<p>Defines the connection to the external database or file server. This can be specified within a connection string (e.g. <code>'ftp://192.168.1.1/'</code>) and the corresponding login information.</p> <p>For regular ETL jobs you can also take use of connections where the connection data like user name and password can easily be encapsulated. For details please refer to CREATE CONNECTION in Section 2.2.3, “Access control using SQL (DCL)”.</p> <p>The declaration of user and password within the IMPORT command are optional. If they are omitted, the data of the connection string or the connection object are used. For JDBC connections, it is possible to use Kerberos authentication by specifying specific data in the IDENTIFIED BY field. This data consists of a key which indicates that Kerberos authentication should be used (<code>ExaAuthType=Kerberos</code>), a base64 encoded configuration file and a base64 encoded keytab file containing the credentials for the principal. The syntax looks like the following: <code>IMPORT INTO table1 FROM JDBC AT '<JDBC_URL>' USER '<kerberos_principal>' IDENTIFIED BY 'ExaAuthType=Kerberos;<base64_krb_conf>;<base64_keytab>' TABLE table2;</code> Further details and examples can be found in our solution center: https://wwwexasol.com/portal/display/SOL-512</p>

Element	Meaning																		
csv_cols	<p>Defines which and how the columns of the CSV file are written. Please also refer to The CSV Data format.</p> <p>col_nr Defines the column number (starting from 1). Alternatively, you can define a certain column range via .. (e.g. 5..8 for columns 5,6,7,8). Please note that column numbers have to be in ascending order.</p> <p>FORMAT Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models”.</p> <p>DELIMIT In default case (AUTO), the field delimiters are written only if special characters occur within the data: the COLUMN SEPARATOR, the ROW SEPARATOR, the COLUMN DELIMITER or a whitespace character. By using the options ALWAYS or NEVER you can define whether column separators shall be written always or never. This local column option overwrites the global option (see <code>file_opts</code>).</p> <p>Please note that if you use the NEVER option, then it's not guaranteed that the exported data can be imported again into Exasol!</p> <p>In the following example the first 4 columns of the source are loaded into the CSV file, the last column with the specified date format:</p> <pre>(1..3,4 FORMAT='DD-MM-YYYY')</pre>																		
fbv_cols	<p>Defines which and how the columns of the FBV file are interpreted. Please also refer to The Fixblock Data format (FBV).</p> <p>The following elements can be specified:</p> <p>SIZE Defines the number of bytes of the column, default is calculated by the source data type:</p> <table border="1"> <thead> <tr> <th>Data type</th><th>Default number of bytes</th></tr> </thead> <tbody> <tr> <td>BOOL</td><td>1</td></tr> <tr> <td>DECIMAL(p)</td><td>p+1 (Sign)</td></tr> <tr> <td>DECIMAL(p,s)</td><td>p+2 (Sign+Point)</td></tr> <tr> <td>DECIMAL(p,p)</td><td>p+3 (Sign+Point+Leading zero)</td></tr> <tr> <td>DOUBLE</td><td>21</td></tr> <tr> <td>DECIMAL/DOUBLE with format</td><td>Dependent of format string</td></tr> <tr> <td>DATE/TIMESTAMP</td><td>Dependent of format string</td></tr> <tr> <td>VARCHAR(n)/CHAR(n)</td><td>n*4 in case of UTF8 columns, n in case of ASCII columns</td></tr> </tbody> </table> <p>FORMAT Optional format definition for numbers or datetime values (default: session format). Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models”.</p> <p>ALIGN Alignment of the column (LEFT or RIGHT), default is LEFT.</p>	Data type	Default number of bytes	BOOL	1	DECIMAL(p)	p+1 (Sign)	DECIMAL(p,s)	p+2 (Sign+Point)	DECIMAL(p,p)	p+3 (Sign+Point+Leading zero)	DOUBLE	21	DECIMAL/DOUBLE with format	Dependent of format string	DATE/TIMESTAMP	Dependent of format string	VARCHAR(n)/CHAR(n)	n*4 in case of UTF8 columns, n in case of ASCII columns
Data type	Default number of bytes																		
BOOL	1																		
DECIMAL(p)	p+1 (Sign)																		
DECIMAL(p,s)	p+2 (Sign+Point)																		
DECIMAL(p,p)	p+3 (Sign+Point+Leading zero)																		
DOUBLE	21																		
DECIMAL/DOUBLE with format	Dependent of format string																		
DATE/TIMESTAMP	Dependent of format string																		
VARCHAR(n)/CHAR(n)	n*4 in case of UTF8 columns, n in case of ASCII columns																		

Element	Meaning
PAD-DING	<p>Padding characters for columns. In the default case, Space is used. You can specify any ASCII character, either in plain text (e.g.: '+'), as hexadecimal value (e.g: '0x09') or as abbreviation (one of 'NUL','TAB','LF','CR','ESC').</p> <p>In the following example 4 columns of a FBV file are written. The first column is right-aligned and filled to 8 bytes with + characters, the fourth column has the specified date format:</p> <pre>(SIZE=8 PADDING='+' ALIGN=RIGHT, ' ' FORMAT='DD-MM-YYYY')</pre>
file_opts	<p>REPLACE Replaces the target if it already exists.</p> <p> Please consider that in case of HTTP[S] and SFTP servers the target file will always be created newly due to protocol limitations.</p> <p>TRUNCATE Deletes the data of the target before loading data.</p> <p>ENCODING Encoding of the CSV or FBV file (default is UTF8). All supported encodings can be found in Appendix D, Supported Encodings for ETL processes and EXAplus.</p> <p>NULL Representation of NULL values. If you didn't specify that option, NULL values are represented as the empty string.</p> <p>BOOLEAN Representation of boolean values. The following value pairs can be defined in a string:</p> <pre>'1/0', 'TRUE/FALSE', 'true/false', 'True/False', 'T/F', 't/f', 'y/n', 'Y/N', 'yes/no', 'Yes/No', 'YES/NO'</pre> <p> Please note that these pairs are automatically accepted when inserting strings into a boolean column using the IMPORT command.</p> <p>ROW SEPARATOR Line break character: <ul style="list-style-type: none"> • 'LF' (Default): corresponds to the ASCII character 0x0a • 'CR': corresponds to the ASCII character 0x0d • 'CRLF': corresponds to the ASCII characters 0x0d and 0x0a • 'NONE': no line break (only allowed for FBV files) </p> <p>COLUMN SEPARATOR Defines the field separator for CSV files. In the default case, the comma (,) is used. You can specify any string, either as plain text (e.g.: ','), as a hexadecimal value (e.g.:</p>

Element	Meaning
	'0x09') or as an abbreviation (one of 'NUL', 'TAB', 'LF', 'CR', 'ESC'). A plain text value is limited to 10 characters, which will be automatically converted to the file's specified ENCODING (see above). A hexadecimal value is limited to 10 bytes (not characters) and will not be converted.
COLUMN DELIMITER	Defines the field delimiter for CSV files. In the default case, the double quote ("") is used. You can specify any string, either as plain text (e.g.: ""), as a hexadecimal value (e.g.: '0x09') or as an abbreviation (one of 'NUL', 'TAB', 'LF', 'CR', 'ESC'). A plain text value is limited to 10 characters, which will be automatically converted to the file's specified ENCODING (see above). A hexadecimal value is limited to 10 bytes (not characters) and will not be converted. If you don't want to use any field delimiter, you can define the empty string ("") or use the option DELIMIT NEVER (see below).
DELIMIT	In the default case (AUTO), the column delimiter is written only if special characters occur within the data: the COLUMN SEPARATOR, the ROW SEPARATOR, the COLUMN DELIMITER or a whitespace character. By using the options ALWAYS or NEVER you can define whether column separators shall be written always or never. This global option can be overwritten within the single column definitions (see csv_cols). Please note that if you use the NEVER option, then it's not guaranteed that the exported data can be imported again into Exasol!
WITH COLUMN NAMES	By the help of this option (only possible for CSV files), an additional row is written at the beginning of the file which contains the column names of the exported table. In case of a subselect that can also be expressions. The other options like e.g. the column separator are also applied for that row. If you want to import the same file again using the IMPORT statement, you can use the option SKIP 1.
error_clause	This clause defines how many invalid rows of the source are allowed. E.g. in case of REJECT LIMIT 5 the statement would work fine if less or equal than five invalid rows occur, and would throw an exception after the sixth error. REJECT LIMIT 0 has the same behavior as though you omit the error clause completely.

Example(s)

```
EXPORT tab1 INTO CSV
  AT 'ftp://192.168.1.1/' USER 'agent_007' IDENTIFIED BY 'secret'
  FILE 'tab1.csv'
  COLUMN SEPARATOR = ';'
  ENCODING = 'Latin1'
  WITH COLUMN NAMES;

CREATE CONNECTION my_connection
```

```
TO 'ftp://192.168.1.1/' USER 'agent_007' IDENTIFIED BY 'secret';

EXPORT (SELECT * FROM T WHERE id=3295) INTO FBV
AT my_connection
FILE 't1.fbv' FILE 't2.fbv'
REPLACE;

EXPORT (SELECT * FROM my_view) INTO EXA
AT '192.168.6.11..14:8563'
USER 'my_user' IDENTIFIED BY 'my_secret'
TABLE my_schema.my_table
CREATED BY 'CREATE TABLE my_table(order_id INT, price DEC(18,2))';

EXPORT tab1 INTO JDBC
AT 'jdbc:exa:192.168.6.11..14:8563'
USER 'agent_007' IDENTIFIED BY 'secret'
TABLE my_schema.tab1;

EXPORT tab1 INTO CSV
AT 'http://HadoopNode:50070/webhdfs/v1/tmp'
FILE 'file.csv?op=CREATE&user.name=user';

EXPORT tab1 INTO CSV
AT 'https://testbucket.s3.amazonaws.com'
USER '<AccessKeyID>' IDENTIFIED BY '<SecretAccessKey>'
FILE 'file.csv';

EXPORT tab1 INTO SCRIPT etl.export_hcat_table
WITH HCAT_DB      = 'default'
      HCAT_TABLE   = 'my_hcat_table'
      HCAT_ADDRESS = 'hcatalog-server:50111'
      HDFS_USER    = 'hdfs';

EXPORT tab1 INTO LOCAL CSV FILE '~/my_table.csv'
COLUMN SEPARATOR = ';' SKIP = 5;
```

2.2.3. Access control using SQL (DCL)

The SQL statements of the Data Control Language (DCL) are used to control the database access rights. Through the management of users and roles as well as the granting of privileges, it can be determined who is permitted to perform what actions in the database.

An introduction to the basic concepts of rights management can be found in [Section 3.2, “Rights management”](#). Further details such as a list of all privileges, a summary of the access rights for SQL statements as well as the system tables relevant to rights management are set out in [Appendix B, Details on rights management](#).

In addition, within the SQL reference for each SQL statement, the prerequisites in terms of which privileges are necessary for the respective statement are specified.

CREATE USER

Purpose

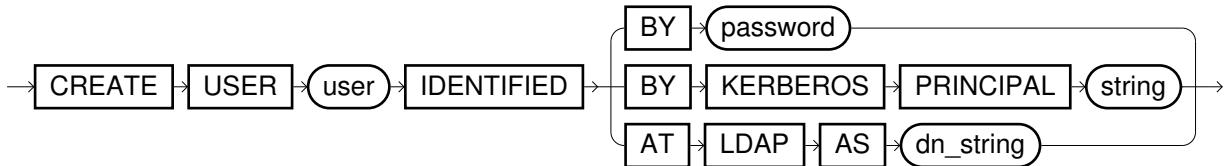
Adds a user to the database. Exasol uses the specified password for authentication.

Prerequisite(s)

- The system privilege CREATE USER

Syntax

create_user::=



Note(s)

- In order for the user to be able to login subsequently, the system privilege CREATE SESSION must also be granted.
- For the user name, the same rules as for SQL identifiers (see [Section 2.1.2, “SQL identifier”](#)) apply. However, even with identifiers in quotation marks no attention is paid to case sensitivity. This means that the usernames "Test", "TEST" and test are synonymous.
- You can choose one of two authentication methods:

Via Password

During the log in, the database checks the given password directly. Please consider that passwords have to be specified as identifiers (see [Section 2.1.2, “SQL identifier”](#)). If you use delimited (quoted) identifiers, then the password is case sensitive.



In case of a regular identifier (unquoted) the password will be set to uppercase letters and has to be adjusted for the login.

Via Kerberos

The drivers authenticate via Kerberos service (single sign-on). Typically, the defined principal looks like the following: <user>@<realm>. Additional information about the overall Kerberos configuration can be found in the corresponding driver chapters ([Chapter 4, Clients and interfaces](#)) and in our Operational Manual: <https://wwwexasol.com/portal/display/DOC/Operational+Manual>

Via LDAP

The database checks the password against a **LDAP** server which can be configured per database within EXAoperation. The parameter *dn-string* (string in single quotes) specifies the so called *distinguished name* which is the user name configured in the **LDAP** server. Not supported are **SASL** and a certification management.

- After creation of the user, no schema exists for this user.
- A user can be renamed by the command **RENAME**.

Example(s)

```
CREATE USER user_1 IDENTIFIED BY "h12_xhz";
CREATE USER user_2 IDENTIFIED AT LDAP
AS 'cn=user_2,dc=authorization,dc=exasol,dc=com';
```

ALTER USER**Purpose**

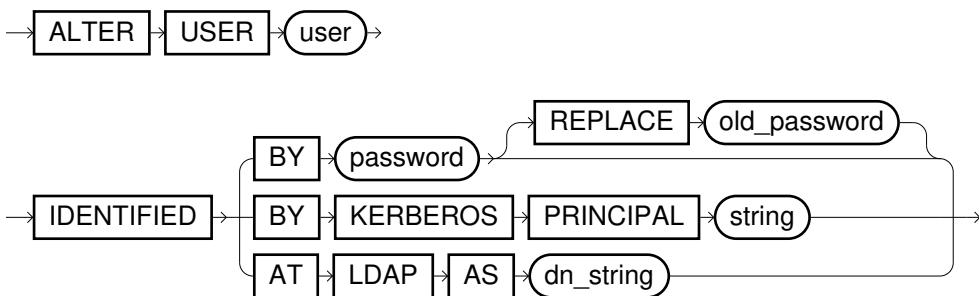
Changes the password of a user.

Prerequisite(s)

- If password authentication is set, then a customer can always change the own password.
- Setting a new password for other users or defining the Kerberos / LDAP authentication needs the system privilege **ALTER USER**.

Syntax

`alter_user ::=`

**Note(s)**

- If one possesses the system privilege **ALTER USER**, the **REPLACE** clause is optional and the old password is not verified.
- For security reasons, the old password must also be specified if a user wishes to change his own password (unless he possesses the system privilege, **ALTER USER**).
- Details to the Kerberos / LDAP authentication and the rules for password creation can be found at **CREATE USER**.

Example(s)

```
ALTER USER user_1 IDENTIFIED BY "h22_xhz" REPLACE "h12_xhz";
-- ALTER_USER privilege necessary for next commands
ALTER USER user_1 IDENTIFIED BY "h12_xhz";
```

```
ALTER USER user_2 IDENTIFIED AT LDAP  
AS 'cn=user_2,dc=authorization,dc=exasol,dc=com';
```

DROP USER

Purpose

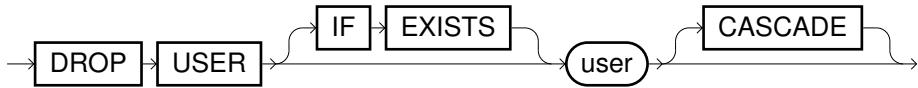
Deletes a user as well as the schemas of that user including all of the schema objects contained therein.

Prerequisite(s)

- The system privilege DROP USER.

Syntax

drop_user ::=



Note(s)

- If CASCADE is specified, all of the schemas of the user as well as their contents will be deleted! Furthermore, all foreign keys which reference the tables of the user are deleted database-wide.
- If schemas that belong to the user still exist, CASCADE must be specified or these must be explicitly deleted beforehand (using DROP SCHEMA).
- If the optional IF EXISTS clause is specified, then the statement does not throw an exception if the user does not exist.
- If the user to be deleted is logged-in at the same time, an error message is thrown and the user is not dropped. In this case it is recommended to revoke the CREATE SESSION privilege from this user and terminate its session using the command [KILL](#).

Example(s)

```
DROP USER test_user1;  
DROP USER test_user2 CASCADE;
```

CREATE ROLE

Purpose

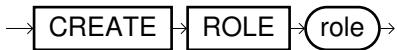
Creates a role.

Prerequisite(s)

- The system privilege CREATE ROLE.

Syntax

create_role ::=



Note(s)

- A role possesses no privileges after creation. These are assigned with the GRANT statement. A role is either granted privileges directly or other roles are assigned to it.
- The same rules apply for role names as for usernames (see [CREATE USER](#)).
- A role can be renamed by the command [RENAME](#).

Example(s)

```
CREATE ROLE test_role;
```

DROP ROLE

Purpose

Deletes a role.

Prerequisite(s)

- Either the system privilege DROP ANY ROLE or this role with the WITH ADMIN OPTION must be assigned to the user.

Syntax

drop_role ::=



Note(s)

- If CASCADE is specified, all of the schemas of the role as well as their contents will be deleted!
- If schemas that belong to the role still exist, CASCADE must be specified or these must be explicitly deleted beforehand (using DROP SCHEMA).
- This statement will also remove the role from other users who possessed it. However, open transactions of such users are not affected.
- If the optional IF EXISTS clause is specified, then the statement does not throw an exception if the role does not exist.
- If you have created a role that doesn't mean the you can delete it.

Example(s)

```
DROP ROLE test_role;
```

CREATE CONNECTION

Purpose

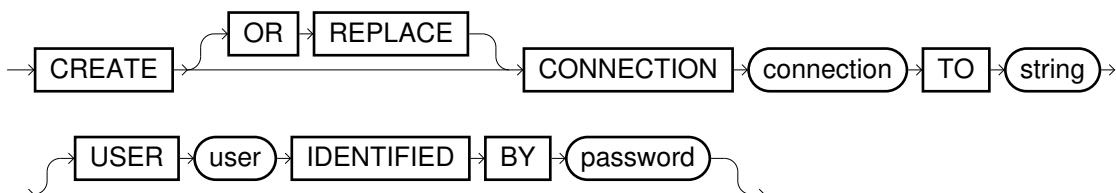
Creates an external connection.

Prerequisite(s)

- System privilege CREATE CONNECTION
- If the OR REPLACE option has been specified and the connection already exists, the rights for [DROP CONNECTION](#) are also needed.

Syntax

create_connection ::=



Note(s)

- External connections can be used within the statements [IMPORT](#) and [EXPORT](#). Users must have the corresponding access rights to the connection (via [GRANT](#)). The connection is automatically granted to the creator (including the ADMIN OPTION).
- Further, connections can control the read access for users who want to use scripts for processing data from local buckets stored in BucketFS. Details can be found in [Section 3.6.4, “The synchronous cluster file system BucketFS”](#).
- You can define an Exasol connection, a native connection to an Oracle database or a JDBC connection to any database. Some JDBC drivers are already delivered as default (visible in EXAoperation) and can be addressed within the connection string (e.g. `jdbc:mysql`, `jdbc:postgres`). You can additionally configure JDBC drivers in EXAoperation and choose them via the DRIVER option if its prefix is ambiguous.

Only the pre-installed JDBC drivers (marked gray in EXAoperation) are tested and officially supported. But our support will try to help you in case of problems with other drivers.

- The declaration of user and password is optional and can be specified within the [IMPORT](#) and [EXPORT](#) statements.
- Invalid connection data will not be noticed before the usage within the [IMPORT](#) and [EXPORT](#) statements.
- The list of all database connections can be found in the system table [EXA_DBA_CONNECTIONS](#) (see [Appendix A, System tables](#)).
- A connection can be renamed by the command [RENAME](#).

Example(s)

```

CREATE CONNECTION ftp_connection
  TO 'ftp://192.168.1.1/'
  USER 'agent_007'
  IDENTIFIED BY 'secret';

CREATE CONNECTION exa_connection TO '192.168.6.11..14:8563';
  
```

```

CREATE CONNECTION ora_connection TO '(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.6.54)(PORT = 1521))
  (CONNECT_DATA = (SERVER = DEDICATED)(SERVICE_NAME = orcl)))';

CREATE CONNECTION jdbc_connection_1
  TO 'jdbc:mysql://192.168.6.1/my_user';
CREATE CONNECTION jdbc_connection_2
  TO 'jdbc:postgresql://192.168.6.2:5432/my_db?stringtype=unspecified';

```

ALTER CONNECTION

Purpose

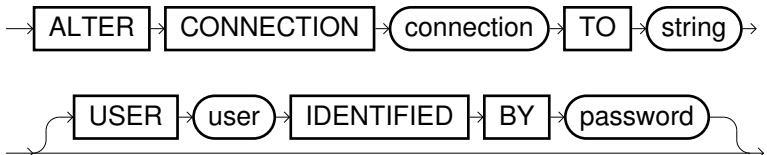
Changes the connection data of an external connection.

Prerequisite(s)

- System privilege ALTER ANY CONNECTION or the connection must be granted to the user with the WITH ADMIN OPTION.

Syntax

alter_connection::=



Example(s)

```

ALTER CONNECTION ftp_connection
  TO 'ftp://192.168.1.1/'
  USER 'agent_008'
  IDENTIFIED BY 'secret';

ALTER CONNECTION exa_connection TO '192.168.6.11..14:8564';

ALTER CONNECTION ora_connection TO '(DESCRIPTION =
  (ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.6.54)(PORT = 1522))
  (CONNECT_DATA = (SERVER = DEDICATED)(SERVICE_NAME = orcl)))';

```

DROP CONNECTION

Purpose

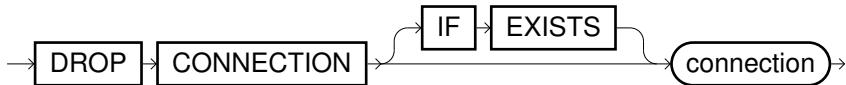
Drops an external connection.

Prerequisite(s)

- System privilege DROP ANY CONNECTION or the connection must be granted to the user with the WITH ADMIN OPTION.

Syntax

drop_connection ::=



Note(s)

- If the optional IF EXISTS clause is specified, then the statement does not throw an exception if the connection does not exist.

Example(s)

```
DROP CONNECTION my_connection;
```

GRANT

Purpose

The GRANT statement can be used to grant system privileges, object privileges, roles or the access to connections to users or roles.

Prerequisite(s)

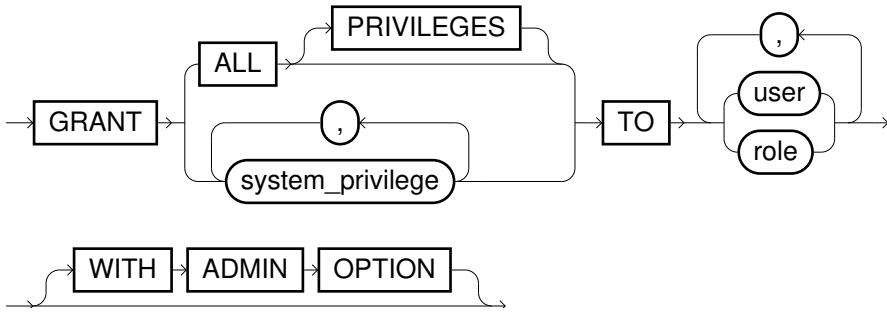
- For system privileges the grantor requires the GRANT ANY PRIVILEGE system privilege or the user must have received this system privilege with the WITH ADMIN OPTION.
- For object rights the grantor must either be the owner of the object or possess the GRANT ANY OBJECT PRIVILEGE system privilege.

 With regard to GRANT SELECT on views, attention must be paid that the grantor is permitted to grant the SELECT on the view and that the owner of the view possesses corresponding SELECT privileges on the base tables, which are grantable to other users by the owner of the view. This is true if either he is the owner of the base tables or possesses the privilege GRANT ANY OBJECT PRIVILEGE. Otherwise, it would be possible to allow any user access to a foreign table by creating a view.

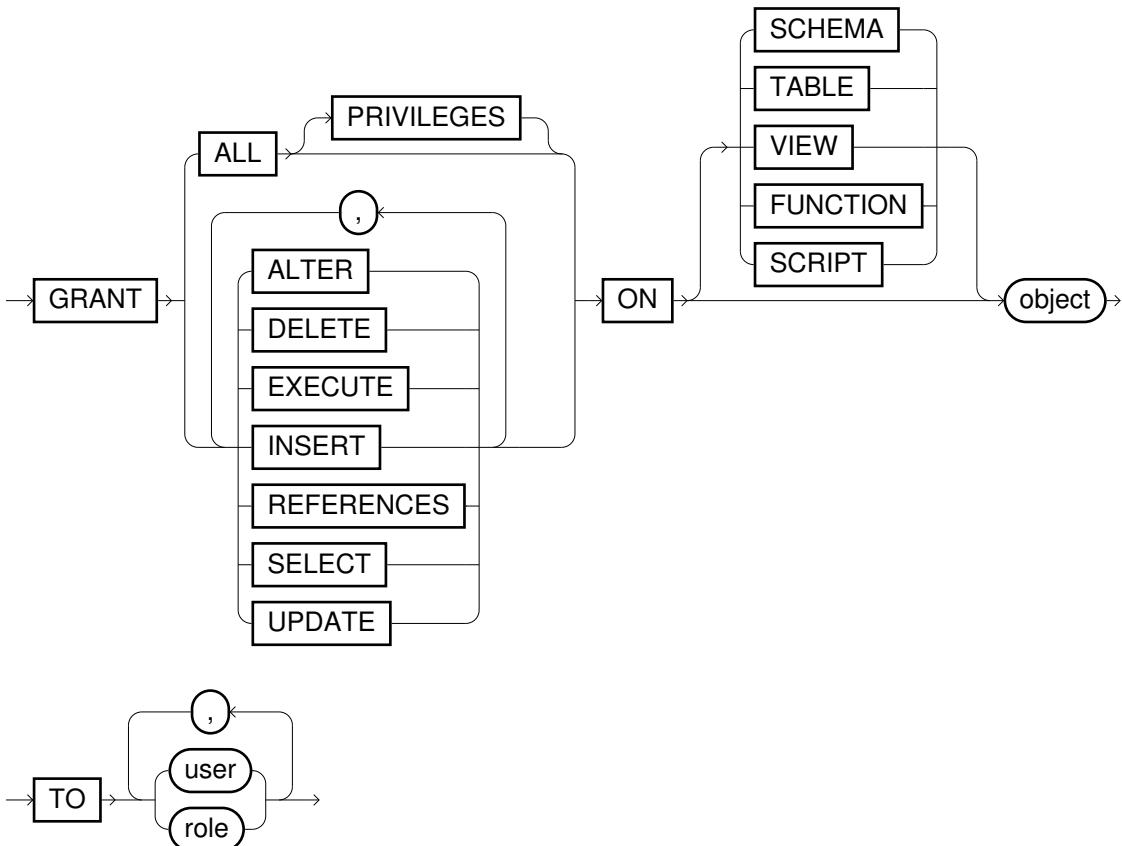
- For roles the grantor requires the GRANT ANY ROLE system privilege or he must have received the role with the WITH ADMIN OPTION.
- For priorities the grantor requires the GRANT ANY PRIORITY system privilege.
- For connections the grantor requires the GRANT ANY CONNECTION system privilege or he must have received the connection with the WITH ADMIN OPTION.

Syntax

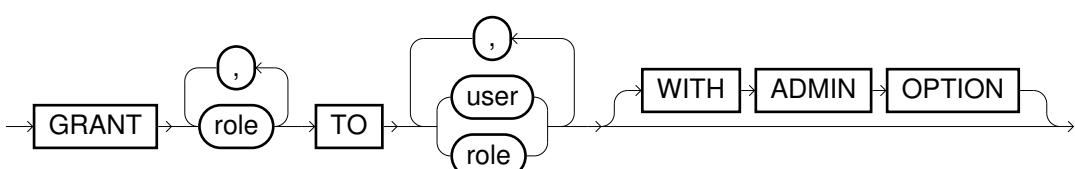
grant_system_privileges ::=



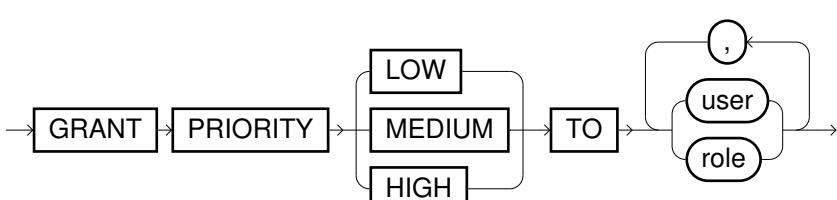
grant_object_privileges ::=



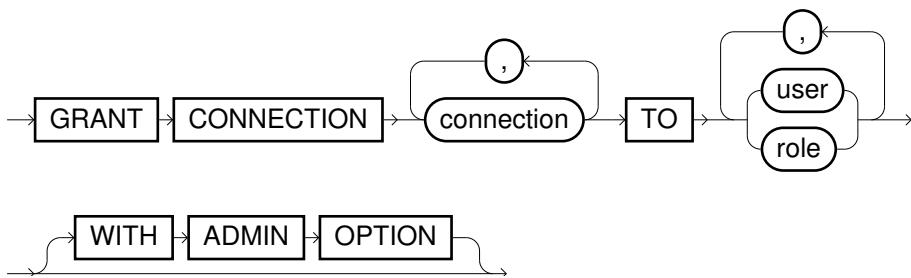
grant_roles ::=



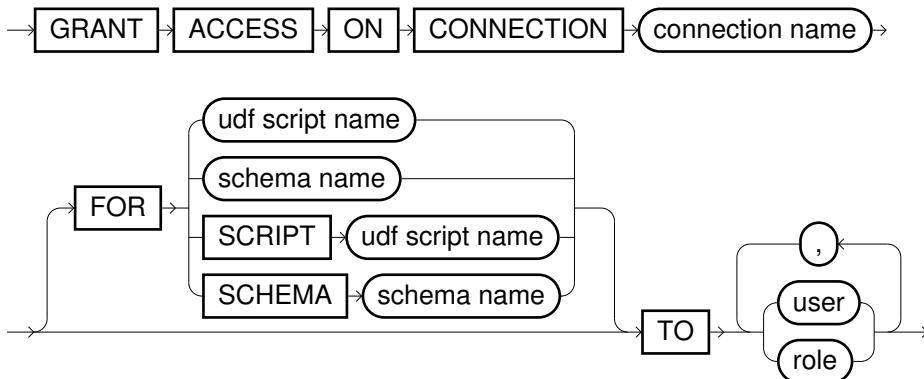
grant_priority ::=



grant_connection ::=



grant_connection_restricted ::=



Note(s)

- The list of system privileges supported by Exasol can be found in [Table B.1, “System privileges in Exasol”](#).
- In order to ensure the security of the database, the GRANT statement should only be used in a very targeted manner. Some of the privileges and roles lead to full control of the database. The DBA role possesses all possible system privileges with the ADMIN option. With the privilege, GRANT ANY PRIVILEGE, it is possible to grant all system privileges. With the ALTER USER privilege it is possible to change the password of SYS. And with the GRANT ANY ROLE privilege it is possible to grant all roles (i.e. also the role DBA for example).
- With GRANT ALL the user is granted all system and object privileges.
- When granting an object privilege to a schema, this privilege is applied to all contained schema objects. Object privileges for virtual schemas and its contained tables is not possible.
- The object privilege REFERENCES cannot be granted to a role.
- Assigned roles cannot be activated or deactivated by the user.
- The ACCESS privilege grants access to the details of a connection (also the password!) for (certain) UDF scripts. This is necessary for adapter scripts of virtual schemas (see also [Section 3.7, “Virtual schemas”](#)).
- Details about priorities and connections can be found in [Section 3.3, “Priorities”](#) and in the descriptions of the statement [CREATE CONNECTION](#).

Example(s)

```

-- System privilege
GRANT CREATE SCHEMA TO role1;
GRANT SELECT ANY TABLE TO user1 WITH ADMIN OPTION;

-- Object privileges
GRANT INSERT ON my_schema.my_table TO user1, role2;
GRANT SELECT ON VIEW my_schema.my_view TO user1;

-- Access on my_view for all users

```

```

GRANT SELECT ON my_schema.my_view TO PUBLIC;

-- Roles
GRANT role1 TO user1, user2 WITH ADMIN OPTION;
GRANT role2 TO role1;

-- Priority
GRANT PRIORITY HIGH TO role1;

-- Connection
GRANT CONNECTION my_connection TO user1;

-- Access to connection details for certain script
GRANT ACCESS ON CONNECTION my_connection
FOR SCRIPT script1 TO user1;

```

REVOKE

Purpose

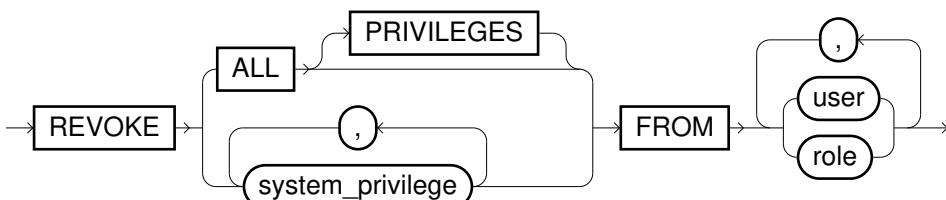
The REVOKE statement can be used to withdraw system privileges, object privileges, roles or the access to connections.

Prerequisite(s)

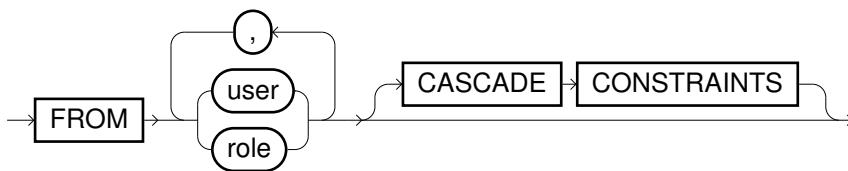
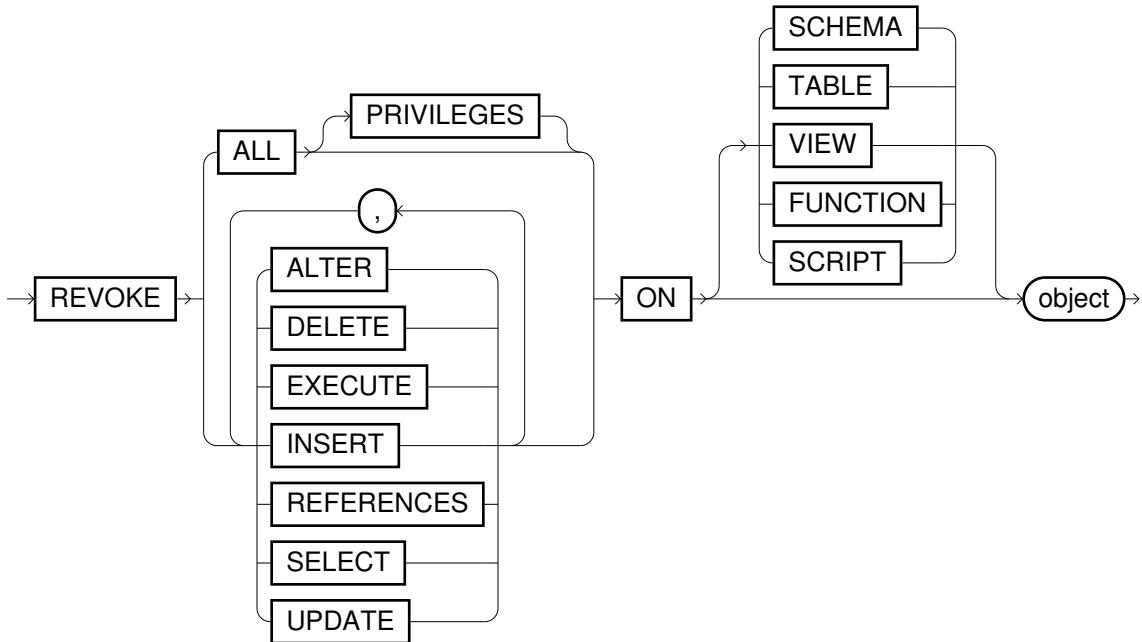
- For system privileges the revoker requires the GRANT ANY PRIVILEGE system privilege or he must have received this system privilege with the WITH ADMIN OPTION.
- For object privileges the revoker requires the GRANT ANY OBJECT PRIVILEGE system privilege or he must be the owner of the object.
- For roles the revoker requires the GRANT ANY ROLE system privilege or he must have received the role with the WITH ADMIN OPTION.
- For priorities the revoker requires the GRANT ANY PRIORITY system privilege.
- For connections the revoker requires the GRANT ANY CONNECTION system privilege or he must have received the connection with the WITH ADMIN OPTION.

Syntax

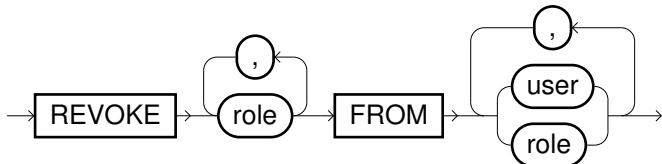
revoke_system_privileges ::=



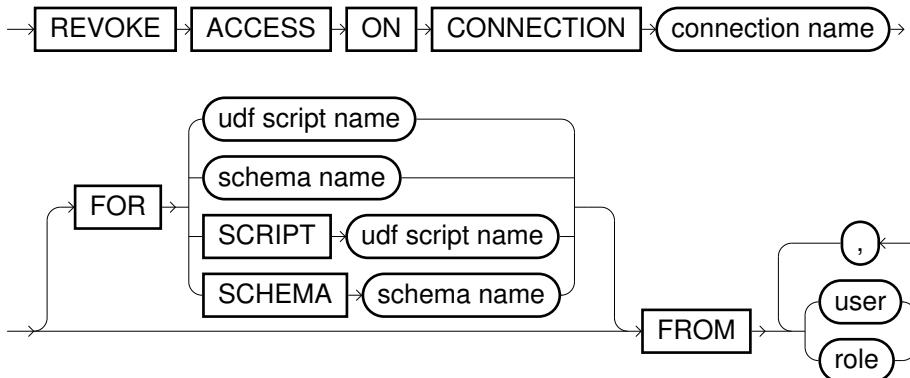
revoke_object_privileges ::=



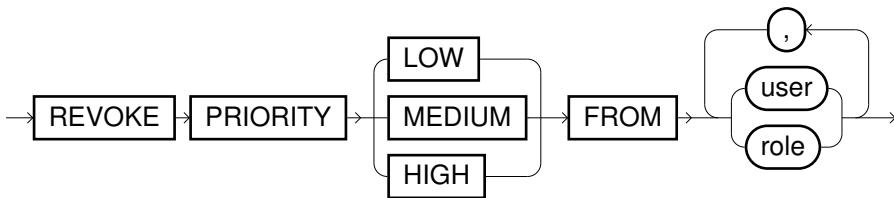
revoke_roles ::=



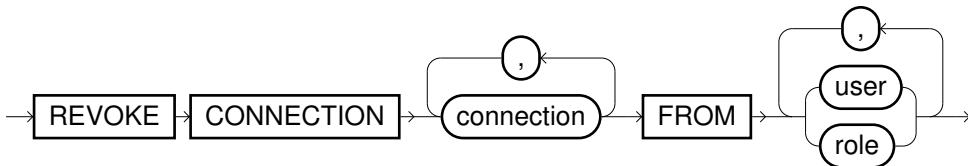
revoke_connection_restricted ::=



revoke_priority ::=



revoke_connections::=



Note(s)

- If the user has received the same privilege or the same role from several users, a corresponding REVOKE will delete all of these.
- If an object privilege was granted to a single schema object, but also to its schema (that means implicitly to all contained objects), and the privilege of the schema was revoked again, then the object privilege for the single schema object is still retained.
- The object privilege REFERENCES can only be revoked if the corresponding user has not yet created foreign keys on that table. In this case you can automatically drop those foreign keys by specifying the option CASCADE CONSTRAINTS.
- As opposed to Oracle, REVOKE ALL [PRIVILEGES] will delete all system or object privileges, even if the user was not granted all rights beforehand by means of GRANT ALL.

Example(s)

```

-- System privilege
REVOKE CREATE SCHEMA FROM role1;

-- Object privileges
REVOKE SELECT, INSERT ON my_schema.my_table FROM user1, role2;
REVOKE ALL PRIVILEGES ON VIEW my_schema.my_view FROM PUBLIC;

-- Role
REVOKE role1 FROM user1, user2;

-- Priority
REVOKE PRIORITY FROM role1;

-- Connections
REVOKE CONNECTION my_connection FROM user1;
  
```

2.2.4. Query language (DQL)

Contents from the database can be queried and analyzed with the Data Query Language (DQL).

SELECT

Purpose

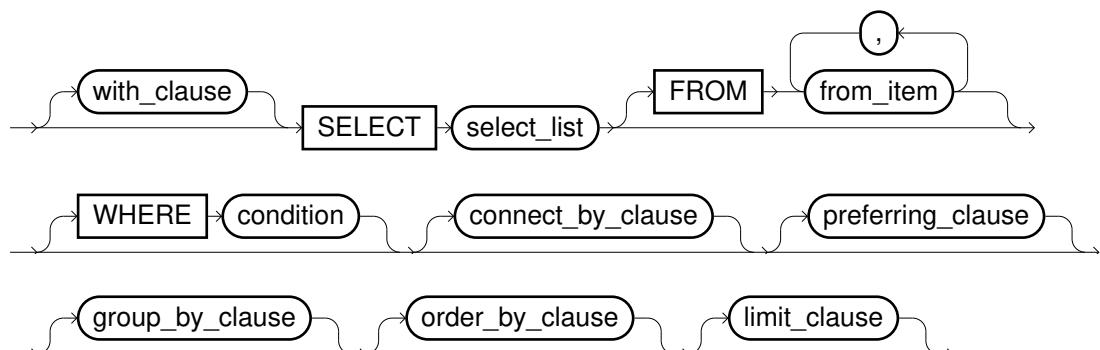
The SELECT statement can be used to retrieve data from tables or views.

Prerequisite(s)

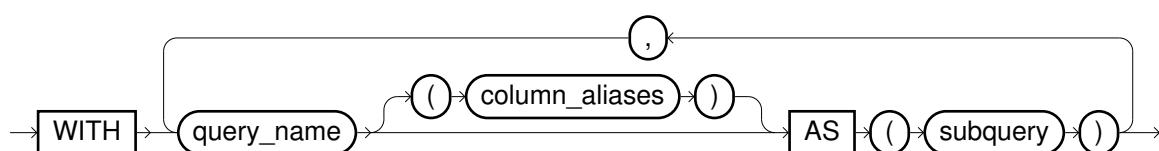
- System privilege `SELECT ANY TABLE` or appropriate `SELECT` privileges on tables or views which are referenced in the `SELECT` list. Either the tables or views belong to the actual user or one of its roles or the actual user owns the object privilege `SELECT` on the table/view or its schema.
 - When accessing views, it is necessary that the owner of the view has appropriate `SELECT` privileges on the referenced objects of the view.
 - If you use a subimport, you need the appropriate rights similar to the `IMPORT` statement.

Syntax

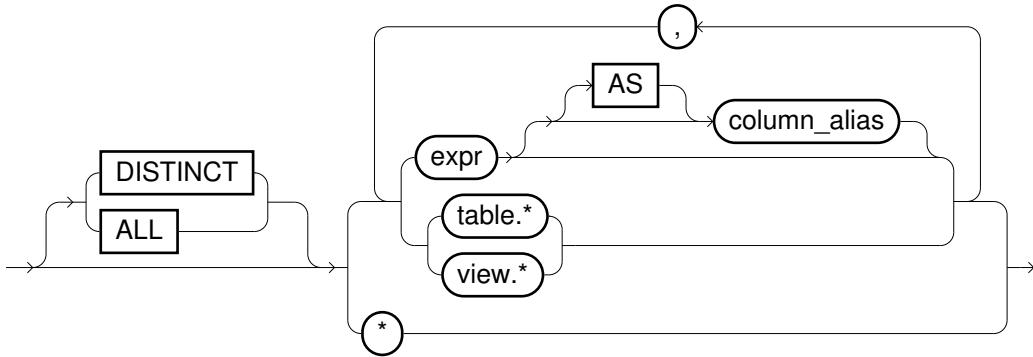
subquery::=



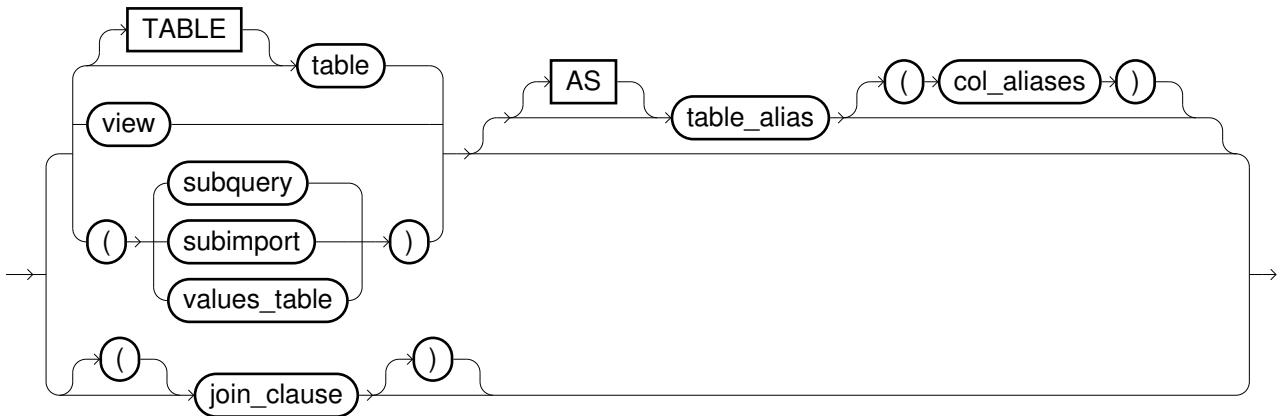
with clause ::=



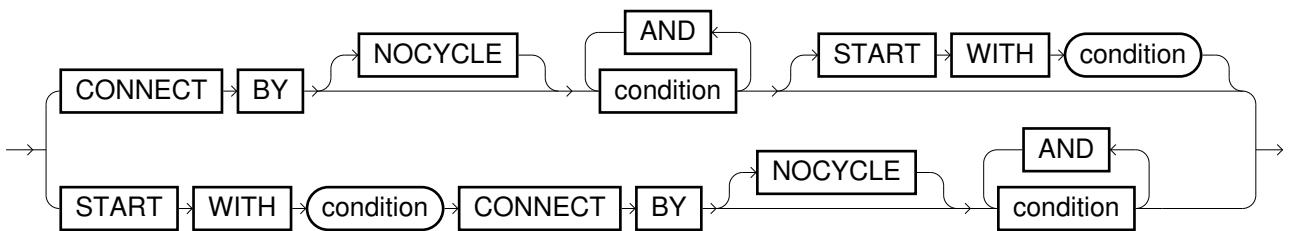
select list::=



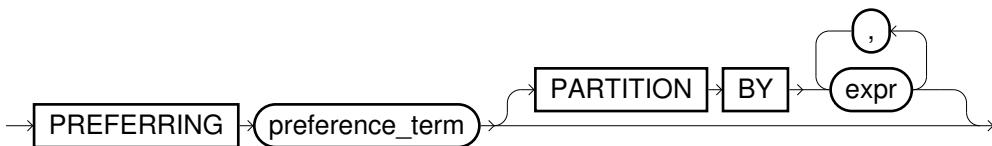
`from_item ::=`



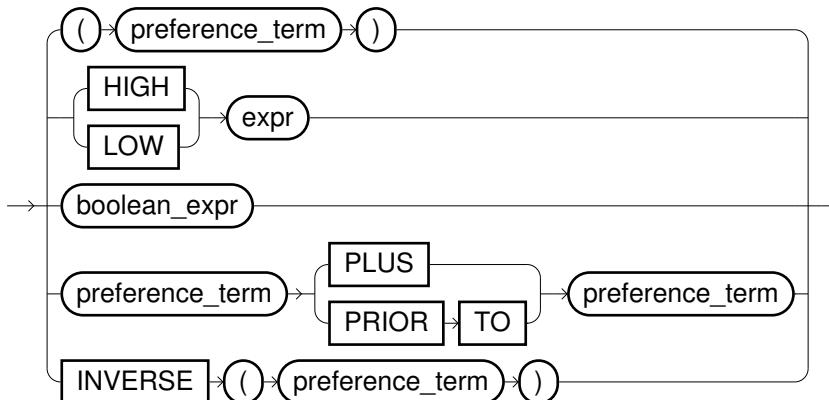
`connect_by_clause ::=`



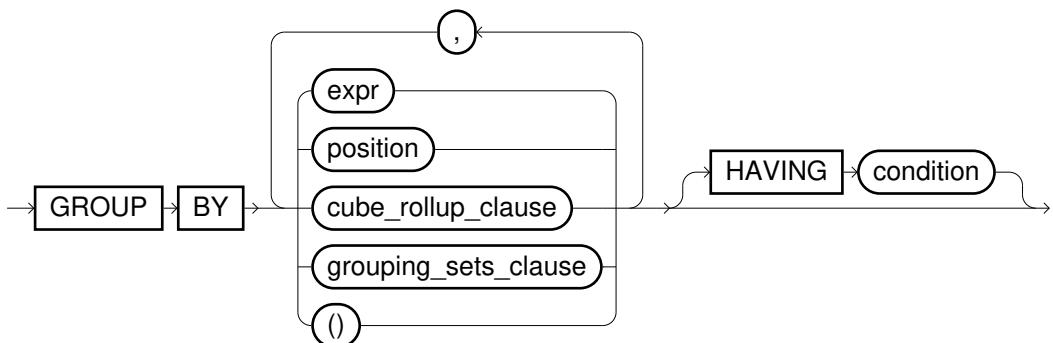
`preferring_clause ::=`



`preference_term ::=`



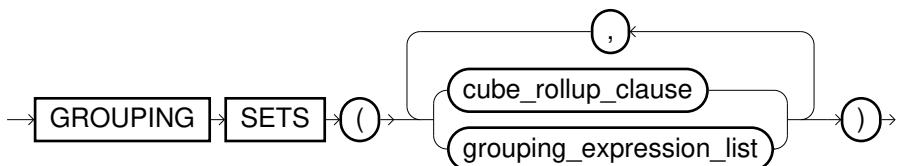
group_by_clause ::=



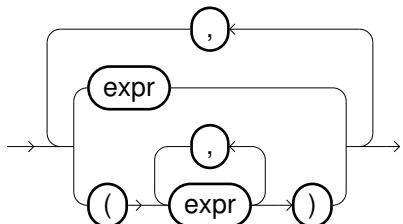
cube_rollup_clause ::=



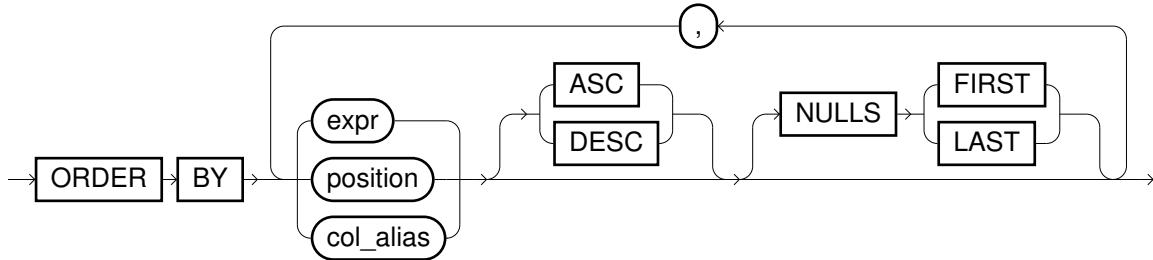
grouping_sets_clause ::=



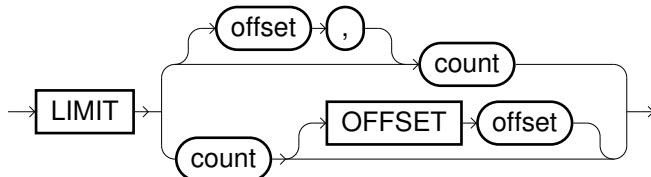
grouping_expression_list ::=



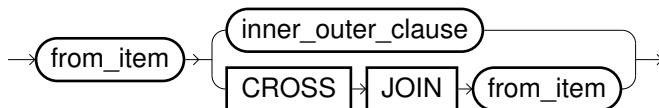
order_by_clause ::=



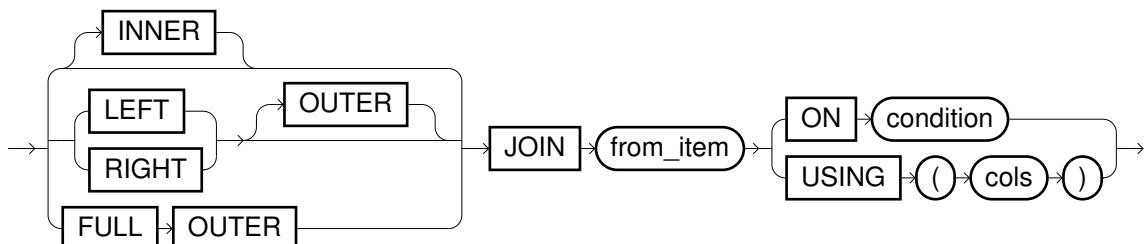
limit_clause ::=



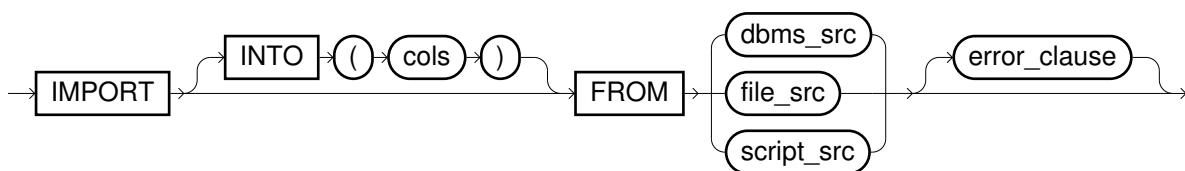
join_clause ::=



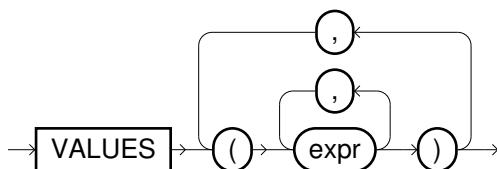
inner_outer_clause ::=



subimport ::=



values_table ::=

**Note(s)**

- You can calculate scalar expressions by omitting the FROM clause (e.g. `SELECT 'abc'`).
- Using the WITH clause you can define temporary views which are only valid during the execution of a subquery.

- In case of DISTINCT, identical rows will be eliminated. If you use the keyword ALL (default), all rows will be present in the result table.
- Source tables and views are defined in the FROM clause. Through the values_table you can easily define static tables, e.g. by (VALUES (1, TRUE), (2, FALSE), (3, NULL)) AS t(i, b) a table with two columns and three rows is specified.
- The SELECT list defines the columns of the result table. If * is used, then all columns will be listed.
- In complex expressions within the SELECT list, the usage of column aliases can be very useful. Directly, such aliases can only be referenced inside the ORDER BY clause. But you can also reference those aliases indirectly via the keyword LOCAL within the other clauses (WHERE, GROUP BY, HAVING) and even in the SELECT list. An example for the indirect referencing: **SELECT ABS(x) AS x FROM t WHERE local.x>10**. Moreover, column aliases define the column names of the result table.
- The WHERE clause can be used to restrict the result by certain filter conditions.
- Equality join conditions among tables can be specified within the WHERE clause by using the = operator. If you want to define an outer condition, add (+) after the outer-expression. This kind of syntax is more readable than using the join_clause.
- You can use the USING clause within a join if the corresponding column names are identical in both tables. In that case, you simply specify the list of unqualified column names which shall be joined, i.e. without specifying table names or aliases. Please also mention that a coalesce expression is used in case of an outer join. Hence, in case of non-matching values, the value is returned and not NULL. Afterwards, only this calculated column can be reference, but not the original columns of the two tables.
- The CONNECT BY clause can be used to define hierarchical conditions. It is evaluated before the WHERE conditions - except the join conditions on tables which are referenced in the CONNECT BY clause.

The following elements are relevant for the CONNECT BY clause (a detailed example can be found below):

START WITH	By this condition you can specify the set of root nodes in the graph.
condition(s)	You can define several conditions. A hierarchical connection between father and son rows can be defined via the keyword PRIOR (e.g. PRIOR employee_id = manager_id). If you don't specify such a PRIOR condition, the cross product will be computed. Thus, the statement <code>SELECT LEVEL FROM dual CONNECT BY LEVEL<=100</code> results in a table with 100 rows, because 100 cross products were calculated for the table dual.
NOCYCLE	If you specify this option, the query also returns results if there exists a cycle. In this case the expansion will be terminated when a cycle is detected.

The following functions and operators can be used in the SELECT list and the WHERE conditions to qualify the results of a hierarchical query.

SYS_CONNECT_BY_PATH (expr, char)	Returns a string containing the full path from the root node to the current node, containing the values for expr and separated by char.
LEVEL	Returns the hierarchy level of a row, i.e. 1 for the root node, 2 for its direct sons, and so on.
PRIOR	References the row's parent row. By that, you can define the father-son condition. But furthermore, you can use this operator to access the values of the parent row. Hence, the following two CONNECT BY conditions are equivalent: <ul style="list-style-type: none"> • PRIOR employee_id = manager_id AND PRIOR employee_id=10. • PRIOR employee_id = manager_id AND manager_id=10.
CONNECT_BY_ROOT	Instead of a row's value, the corresponding value of the root node is used (e.g. CONNECT_BY_ROOT last_name would be evaluated by the name of the highest manager of an employee if the condition PRIOR employee_id = manager_id was defined in the CONNECT BY clause).

CONNECT_BY_ISLEAF

This expression returns 1 if a row is a leaf within the tree (i.e. it has no sons), otherwise 0.

CONNECT_BY_ISCYCLE

Returns whether the current row causes a cycle. In the path (see above) such a row will occur exactly twice. This expression can only be used in combination with the NOCYCLE option.



Please mention that this behavior is different to Oracle's implementation where the parent node is returned instead of the connection point of the cycle.



The function LEVEL and the operator PRIOR can also be used within the CONNECT BY clause.

- The PREFERING clause defines an Skyline preference term. Details can be found in [Section 3.10, “Skyline”](#).
- The GROUP BY clause defines groups of rows which will be aggregated in the result table. Inside the SELECT list, you can use aggregate functions. Using an numerical value x (position) results in aggregating the result table by the x-th column. If GROUP BY is used, all SELECT list elements have to be aggregated except those which define the grouping keys.
- CUBE, ROLLUP and GROUPING SETS are extensions of the GROUP BY clause for calculating superaggregates. Those are hierarchical aggregation levels like e.g. partial sums for days, months and years and can be computed within one single GROUP BY statement instead of using a UNION of several temporary results.

You can distinguish between **regular result rows** (normal aggregation on the deepest level) and **superaggregate rows**. The total of all arguments results in the normal aggregation, the subsets result in the corresponding superaggregates. You can discern the result row types by using the function [GROUPING\[_ID\]](#).

CUBE

Calculates the aggregation levels for all possible combinations of the arguments (2^n combinations).

Example: Via **CUBE(countries,products)** you can sum up all subtotal revenues of all country/product pairs (*regular result rows*), but additionally the subtotals of each country, the subtotals of each product and the total sum (3 additional *superaggregate rows*).

ROLLUP

Calculates the aggregation levels for the first n, n-1, n-2, ... 0 arguments (overall $n+1$ combinations). The last level corresponds to the total sum.

Example: Via **ROLLUP(year,month,day)** you can sum up all revenues of all single date (*regular result rows*), but additionally for each month of year, for each year and the total sum (*superaggregate rows*).

GROUPING SETS

Calculates the aggregation levels for the specified combinations. CUBE and ROLLUP are special forms of GROUPING SETS and simplify the notation.

()

Is similar to **GROUPING SETS ()** and aggregates the whole table as one single group.

If multiple hierarchical groupings are specified, separated by a comma, then the result is the set of all combinations of partial groupings (cross product). E.g. the expression **ROLLUP(a,b),ROLLUP(x,y)** results in overall 9 combinations. Starting with the subsets **(a,b)**, **(a)**, **()** and **(x,y)**, **(x)**, **()** you get the following combinations: **(a,b,x,y)**, **(a,b,x)**, **(a,b)**, **(a,x,y)**, **(a,x)**, **(a)**, **(x,y)**, **(x)**, **()**.

- By using the HAVING clause you can restrict the number of groups.
- The result table can be sorted by specifying the ORDER BY clause. Using an numerical value x (position) results in sorting the result table by the x-th column.

Options of the ORDER BY clause:

- ASC (Default) means ascending, DESC means descending sorting.

- The `NULLS LAST` (Default) and `NULLS FIRST` option can be used to determine whether `NULL` values are sorted at the end or the beginning.

 String data is sorted by its binary representation.

- The number of result rows can be restricted by defining the `LIMIT` clause. The optional offset can only be used in combination with `ORDER BY`, because otherwise the result would not be deterministic. `LIMIT` is not allowed in aggregated `SELECT` statements and within correlated subqueries of an `EXISTS` predicate.
- By using the `subimport` clause, you can integrate the import of external data sources directly in your query. Please note the following notes:
 - Details about the usage of external data sources and their options can be found in the description of the `IMPORT` statement in [Section 2.2.2, “Manipulation of the database \(DML\)”.](#)
 - It is highly recommended to explicitly specify the target column types (see example below). Otherwise, the column names and data types are chosen in a generic way. For importing files, these types are mandatory.
 - Local files cannot be imported directly within queries.
 - By creating a view, external data sources can be transparently integrated in Exasol as a sort of external tables.
 - Local filter conditions on such imports are not propagated to the source databases. However, you can achieve that by using the `STATEMENT` option.
- `SELECT` statements can be combined using the [Table operators UNION \[ALL\], INTERSECT, MINUS](#).
- Please note that `SELECT` queries can be directly returned from the query cache in case the syntactically equivalent query was already executed before. Details can be found in the notes of command [ALTER SYSTEM](#).

Example(s)

The following examples relate to these tables:

```
SELECT * FROM customers;
```

C_ID	NAME
1	smith
2	jackson

```
SELECT * FROM sales;
```

S_ID	C_ID	PRICE	STORE
1	1	199.99	MUNICH
2	1	9.99	TOKYO
3	2	50.00	MUNICH
4	1	643.59	TOKYO
5	2	2.99	MUNICH
6	2	1516.78	NEW YORK

```
SELECT store, SUM(price) AS volume FROM sales
GROUP BY store ORDER BY store DESC;
```

STORE	VOLUME
TOKYO	653.58
NEW YORK	1516.78
MUNICH	252.98

```
SELECT name, SUM(price) AS volume FROM
customers JOIN sales USING (c_id)
```

```

GROUP BY name ORDER BY name;

NAME      VOLUME
----- -----
jackson   1569.77
smith     853.57

WITH tmp_view AS (SELECT name, price, store FROM customers, sales
                  WHERE customers.c_id=sales.c_id)
SELECT sum(price) AS volume, name, store FROM tmp_view
GROUP BY GROUPING SETS (name,store, ());

VOLUME          NAME      STORE
----- -----
      1569.77 jackson
      853.57 smith
      653.58      TOKYO
      252.98      MUNICH
      1516.78     NEW YORK
      2423.34

SELECT * FROM (IMPORT INTO (v VARCHAR(1))
               FROM EXA AT my_connection
               TABLE sys.dual);

SELECT last_name, employee_id id, manager_id mgr_id,
       CONNECT_BY_ISLEAF leaf, LEVEL,
       LPAD(' ', 2*LEVEL-1)||SYS_CONNECT_BY_PATH(last_name, '/') "PATH"
  FROM employees
 CONNECT BY PRIOR employee_id = manager_id
 START WITH last_name = 'Clark'
 ORDER BY employee_id;

LAST_NAME  ID MGR_ID LEAF LEVEL PATH
----- -----
Clark      10  9    0    1    /Clark
Sandler   11 10    1    2    /Clark/Sandler
Smith     12 10    0    2    /Clark/Smith
Jackson   13 10    0    2    /Clark/Jackson
Taylor    14 10    1    2    /Clark/Taylor
Brown     15 12    1    3    /Clark/Smith/Brown
Jones     16 12    1    3    /Clark/Smith/Jones
Popp      17 12    1    3    /Clark/Smith/Popp
Williams  18 13    1    3    /Clark/Jackson/Williams
Johnson   19 13    1    3    /Clark/Jackson/Johnson

```

Table operators UNION [ALL], INTERSECT, MINUS

Purpose

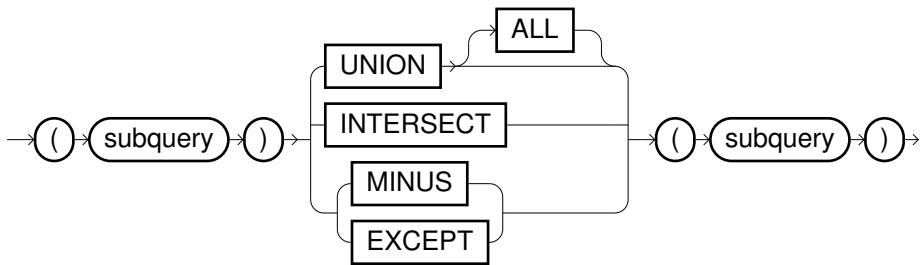
In order to combine the results of various queries with one another, the table operators UNION ALL, UNION, INTERSECT and MINUS (=EXCEPT) exist. These calculate the set union, the set union without duplicates, the intersection without duplicates, and the set difference without duplicates from two subqueries.

UNION ALL Union from both subqueries. All of the rows from both operands are taken into account.

UNION	The set union from both subqueries without duplicates. All of the rows from both operands are taken into account. Duplicate entries in the result are eliminated.
INTERSECT	The intersection from both subqueries without duplicates. All of the rows that appear in both operands are accounted for in the result. Duplicate entries in the result are eliminated.
MINUS or EXCEPT	The set difference from both subqueries without duplicates. The result comprises those rows in the left operand that do not exist in the right operand. Duplicate entries in the result are eliminated.

Syntax

table_operators ::=



Note(s)

- The table operators (except UNION ALL) are expensive operations and can lead to performance problems, in particular with very large tables. This is primarily because the result must not contain duplicates. Removing duplicates is an expensive operation.
- The number of columns of both operands must match and the data types of the columns of both operands must be compatible.
- The names of the left operand are used as columns name for the result.
- Additionally, several table operators can be combined next to one another. In this respect, INTERSECT has higher priority than UNION [ALL] and MINUS. Within UNION [ALL] and MINUS evaluation is performed from left to right. However, for reasons of clarity parentheses should always be used.
- EXCEPT comes from the SQL standard, MINUS is an alias and is, e.g. supported by Oracle. Exasol supports both alternatives.

Example(s)

The following examples relate to these tables:

```
SELECT * FROM t1;
```

I1	C1
1	abc
2	def
3	abc
3	abc
5	xyz

```
SELECT * FROM t2;
```

I2	C2
1	abc
	abc

```
3
4 xyz
4 abc
```

```
(SELECT * from T1) UNION ALL (SELECT * FROM T2);
```

```
I1      C1
-----
1 abc
2 def
3 abc
3 abc
5 xyz
1 abc
abc
3
4 xyz
4 abc
```

```
(SELECT * from T1) UNION (SELECT * FROM T2);
```

```
I1      C1
-----
1 abc
3 abc
4 abc
abc
2 def
4 xyz
5 xyz
3
```

```
(SELECT * from T1) INTERSECT (SELECT * FROM T2);
```

```
I1      C1
-----
1 abc
```

```
(SELECT * from T1) MINUS (SELECT * FROM T2);
```

```
I1      C1
-----
3 abc
2 def
5 xyz
```

2.2.5. Verification of the data quality

The data quality approval statement can be used to analyze and ensure the quality of the data in the database.

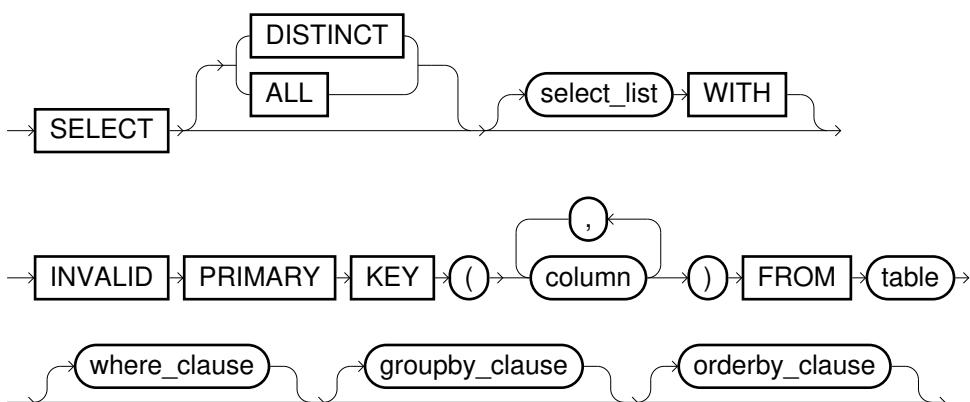
Verification of the primary key property (PRIMARY KEY)

Purpose

This construct can be used to check whether a number of columns have the primary key property. This is the case if the specified columns do not contain duplicates and no NULL values. Rows that do not conform to the primary key property are selected.

Syntax

`select_invalid_primary_key ::=`



Note(s)

- In the formulation without `select_list` invalid rows in the columns to be checked for the primary key property are selected.
- ROWNUM cannot be used in combination with this statement.
- Verification of the primary key property occurs in the table stated in the `FROM` clause. It is not until after this that WHERE, GROUP BY, etc. are used on the table with the columns that violate the property.

Example(s)

The following examples relate to this table:

NR	NAME	FIRST_NAME
1	meiser	inge
2	mueller	hans
3	meyer	karl
3	meyer	karl
5	schmidt	ulla
6		benno
2	fleischer	jan

```
SELECT first_name , name WITH INVALID PRIMARY KEY (nr) from T1;  
  
FIRST_NAME NAME  
----- -----  
hans      mueller  
jan       fleischer  
karl     meyer  
karl     meyer
```

```
SELECT * WITH INVALID PRIMARY KEY (nr,name) from T1;

NR          NAME        FIRST_NAME
-----
      3 meyer      karl
      3 meyer      karl
      6           benno
```

```
SELECT INVALID PRIMARY KEY (first_name) from T1;  
  
FIRST_NAME  
-----  
karl  
karl
```

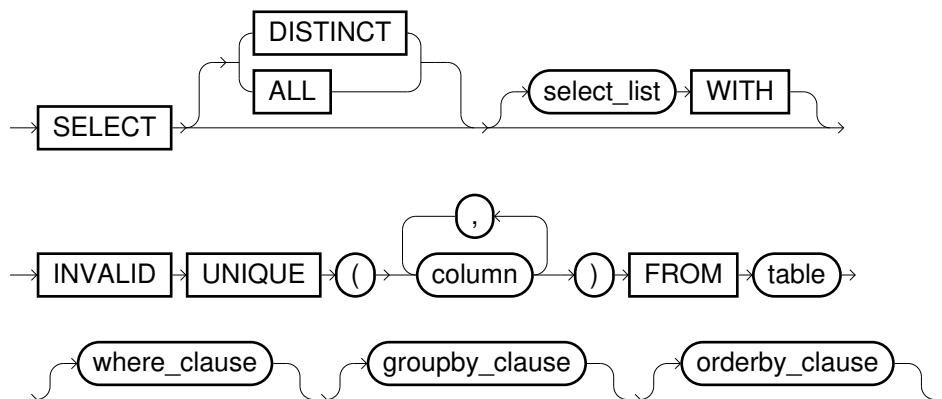
Verification of the uniqueness (UNIQUE)

Purpose

This construct can be used to verify whether the rows of a number of columns `cols` are unique. This is the case if the specified columns `cols` do not contain data records in duplicate. Rows in the specified columns `cols` which only contain NULL values are classified as being unique (even if there is more than one). Non-unique rows are selected.

Syntax

select_invalid_unique::=



Note(s)

- In the formulation without `select_list` invalid rows in the columns to be checked for uniqueness are selected.

- ROWNUM cannot be used in combination with this statement.
- Verification of uniqueness occurs directly in the table specified in the FROM clause. It is not until after this that WHERE, GROUP BY, etc. are used on the table with the columns that violate the uniqueness.

Example(s)

The following examples relate to this table:

```
SELECT * FROM t1;

NR      NAME      FIRST_NAME
-----
1 meiser    inge
2 mueller   hans
3 meyer     karl
3 meyer     karl
5 schmidt   ulla
6           benno
2 fleischer jan
3
```

```
SELECT INVALID UNIQUE (first_name, name) from T1;

FIRST_NAME NAME
-----
karl      meyer
karl      meyer
```

```
SELECT * WITH INVALID UNIQUE (nr,name) from T1;

NR      NAME      FIRST_NAME
-----
3 meyer    karl
3 meyer    karl
```

```
SELECT first_name WITH INVALID UNIQUE (nr, name, first_name) from T1;

FIRST_NAME
-----
karl
karl
```

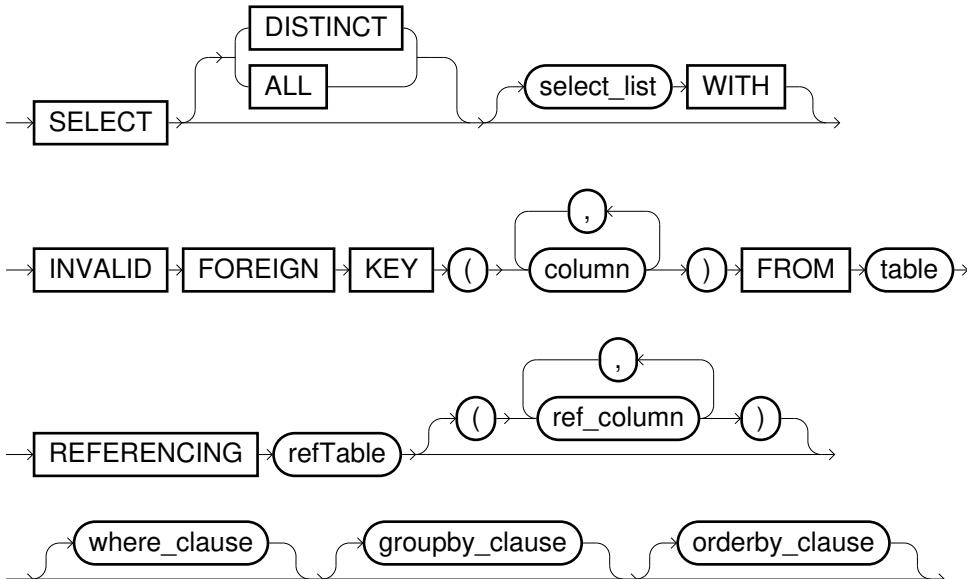
Verification of the foreign key property (FOREIGN KEY)

Purpose

Selects rows that violate the specified foreign key property. This is the case if the row values of the specified columns contain NULL values or do not exist in the specified columns of the referenced table.

Syntax

```
select_invalid_foreign_key ::=
```



Note(s)

- Preferably, the referenced columns of the reference table should possess the primary key property. However, this is not verified by this statement.
- In the formulation without select_list invalid rows in the columns to be checked for the foreign key property are selected.
- ROWNUM cannot be used in combination with this statement.
- Verification of the foreign key property occurs directly in the table specified in the FROM clause. It is not until after this that WHERE, GROUP BY, etc. are used on the table with the columns that violate the property.

Example(s)

The following examples relate to these tables:

```
SELECT * FROM t1;
```

NR	NAME	FIRST_NAME
1	meiser	inge
2	mueller	hans
3	meyer	karl
3	meyer	karl
5	schmidt	ulla
6		benno
2	fleischer	jan

```
SELECT * FROM t2;
```

ID	NAME	FIRST_NAME
1	meiser	otto
2	mueller	hans
3	meyer	karl
5	schmidt	ulla
6		benno
7	fleischer	jan

```
SELECT first_name , name WITH INVALID FOREIGN KEY (nr) from T1  
    REFERENCING T2 (id);  
  
-- Empty result because all of the values of nr exist in the column id  
  
FIRST_NAME NAME  
-----
```

```
SELECT * WITH INVALID FOREIGN KEY (first_name , name) from T1  
    REFERENCING T2;  
  
NR      NAME      FIRST_NAME  
-----  
1      meiser    inge  
6      benno     benno
```

```
SELECT INVALID FOREIGN KEY (nr,first_name , name) from T1  
    REFERENCING T2 (id, first_name , name);  
  
NR      FIRST_NAME NAME  
-----  
1      inge       meiser  
6      benno     benno  
2      jan        fleischer
```

2.2.6. Other statements

The SQL statements that cannot be sorted in one of the previous chapters are explained in this chapter.

COMMIT

Purpose

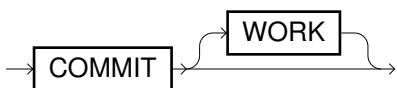
The COMMIT statement is used to persistently store changes of the current transaction in the database.

Prerequisite(s)

- None.

Syntax

commit ::=



Note(s)

- The keyword WORK is optional and is only supported in order to conform to the SQL standard.
- The automatic running of COMMIT after each SQL statement is possible with the EXAplus [SET AUTOCOMMIT](#) command.
- More information on transactions can be found in [Section 3.1, “Transaction management”](#).

Example(s)

```

CREATE TABLE t (i DECIMAL);
INSERT INTO t values (1);
COMMIT;
SELECT COUNT(*) FROM t;

COUNT(*)
-----
1

-- table's data was already persistently committed
ROLLBACK;

SELECT COUNT(*) FROM t;

COUNT(*)
-----
1
  
```

ROLLBACK

Purpose

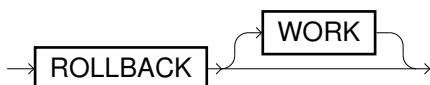
The ROLLBACK statement is used to withdraw changes of the current transaction.

Prerequisite(s)

- None.

Syntax

rollback ::=



Note(s)

- The keyword WORK is optional and is only supported in order to conform to the SQL standard.
- More information on transactions can be found in [Section 3.1, “Transaction management”](#).

Example(s)

```
CREATE TABLE t (i DECIMAL);
COMMIT;

INSERT INTO t values (1);
SELECT COUNT(*) FROM t;

COUNT(*)
-----
1

ROLLBACK;

SELECT COUNT(*) FROM t;

COUNT(*)
-----
0
```

EXECUTE SCRIPT

Purpose

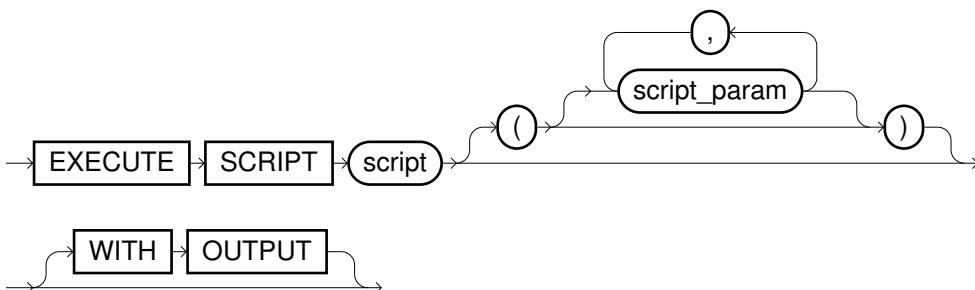
By using this command you can execute a script.

Prerequisite(s)

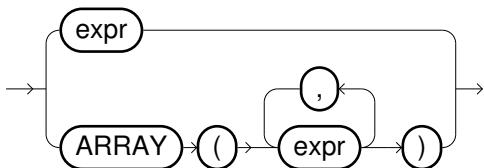
- System privilege EXECUTE ANY SCRIPT, object privilege EXECUTE on the script, or the current user owns the script.

Syntax

execute_script ::=



script_param ::=



Note(s)

- A script can be created and dropped by using the statements [CREATE SCRIPT](#) and [DROP SCRIPT](#).
- An extended introduction to the script language can be found in [Section 3.5, “Scripting”](#).
- Content and parameters of a script are integrated in the corresponding system tables (e.g. `EXA_ALL_SCRIPTS` – see also [Appendix A, System tables](#)).
- The return value of a script can be a table or a rowcount. It is specified as option in the [CREATE SCRIPT](#) statement (see also section [Return value of a script](#) in [Section 3.5, “Scripting”](#)).
- When specifying the option `WITH OUTPUT`, the return value of the script is ignored. In this case always a result table is returned which consists of all debug messages which are created via the function `output()` during the script execution (see also section [Debug output](#) in [Section 3.5, “Scripting”](#)).
- Contrary to views a script is executed with the privileges of the executing user, not with the privileges of the person which created the script via [CREATE SCRIPT](#).

Example(s)

```

EXECUTE SCRIPT script_1;
EXECUTE SCRIPT script_2 (1,3,'ABC');
EXECUTE SCRIPT script_3 (ARRAY(3,4,5));
  
```

KILL

Purpose

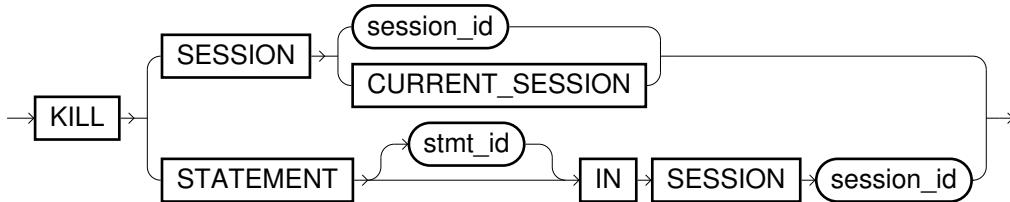
Via this command you can terminate a user session or query.

Prerequisite(s)

- In order to terminate a foreign session or query, the user requires the system privilege `KILL ANY SESSION`.

Syntax

kill::=



Note(s)

- In case of KILL SESSION the corresponding user gets an error message and is logged out of the database.
- KILL STATEMENT terminates the current statement of the corresponding session, but not the session itself. If you specify a certain statement id (see also [EXA_ALL_SESSIONS](#)), you can constrain the termination to a certain statement of a session. If this statement does not exist anymore (e.g. already finished), then a corresponding exception will be thrown. The termination of statements is similar to the Query Timeout (see [ALTER SESSION](#) or [ALTER SYSTEM](#)). When a statement is terminated, it may finish with an exception within a few seconds through an internal cancellation point. If this fails - for example because there are no such cancellation points (e.g. in LUA) or the query is slowed down due to disk operations - the query is terminated forcefully and the transaction is rolled back (including an internal reconnect). If the affected statement is part of [EXECUTE SCRIPT](#), the whole script is terminated.

Example(s)

```
KILL STATEMENT IN SESSION 7792436882684342285
KILL SESSION 7792436882684342285
```

ALTER SESSION

Purpose

This statement can be used to configure the current user session.

Prerequisite(s)

- None.

Syntax

alter_session::=



Note(s)

- The session-based parameters are initialized with the system-wide parameters (see [ALTER SYSTEM](#)), but can be overwritten with the ALTER SESSION statement. The current settings can be found in the [EXA_PARAMETERS](#) system table.
- At the moment that a user logs out, changes to settings made via ALTER SESSION are lost.
- The following parameters can be set:

TIME_ZONE	Defines the time zone in which the values of type TIMESTAMP WITH LOCAL TIME ZONE are interpreted. Further information can be found in section Date/Time data types in Section 2.3, “Data types” . The list of supported timezones can be found in the system table EXA_TIME_ZONES . The function SESSIONTIMEZONE returns the current session time zone.
TIME_ZONE_BEHAVIOR	Defines the course of action for ambiguous and invalid timestamps within a certain time zone. Further information can be found in section Date/Time data types in Section 2.3, “Data types” .
TIMESTAMP_ARITHMETIC_BEHAVIOR	Defines the behavior for +/- operators: <ul style="list-style-type: none"> • INTERVAL - The difference of two datetime values is an interval, and when adding a decimal value to a timestamp, the number is rounded to an integer, so actually a certain number of full days is added. • DOUBLE - The difference of two datetime values is a double, and when adding a decimal value to a timestamp, the fraction of days is added (hours, minutes, ...).
NLS_DATE_FORMAT	Sets the date format used for conversions between dates and strings. Possible formats are described in Section 2.6.1, “Date/Time format models” .
NLS_TIMESTAMP_FORMAT	Sets the timestamp format used for conversions between timestamps and strings. Possible formats are described in Section 2.6.1, “Date/Time format models” .
NLS_DATE_LANGUAGE	Sets the language of the date format used in abbreviated month and day formats and those written in full (see Section 2.6.1, “Date/Time format models”). Possible languages are English (ENG = Default) and German (DEU). The English language can be set using ENG or ENGLISH and the German language with DEU, DEUTSCH, and GERMAN.
NLS_FIRST_DAY_OF_WEEK	Defines the first day of a week (integer 1-7 for Monday-Sunday).
NLS_NUMERIC_CHARACTERS	Defines the decimal and group characters used for representing numbers. This parameter is also relevant to the use of numeric format models (see also Section 2.6.2, “Numeric format models”).
DEFAULT_LIKE_ESCAPE_CHARACTER	Defines the escape character for the LIKE predicate (see Section 2.8, “Predicates”) in case no explicit one was specified.
QUERY_CACHE	The parameter QUERY_CACHE defines the usage of a read cache for SELECT queries. If the syntactically identical query is sent multiple times (except upper/lower case, spaces, ...), then the database can read the result directly out of a cache instead of executing the query. This is only applicable if the corresponding schema objects haven't changed in the meantime. The following values can be set: <ul style="list-style-type: none"> • ON - The query cache is actively used, i.e. query results are read from and written into the cache. • OFF - The query cache is not used. • READONLY - Results are read from the cache, but additional new queries will not be cached.
QUERY_TIMEOUT	Whether a query was returned from the cache can be determined by the column EXECUTION_MODE in the corresponding system tables (e.g. EXA_SQL_LAST_DAY).
	Defines how many seconds a statement may run before it is automatically aborted. When this point is reached, the statement may finish with an exception within a few seconds through an internal cancellation point. If this fails - for example because there are no such cancellation points (e.g. in LUA) or the query is slowed down due to disk operations - the query is terminated forcefully and the transaction is rolled back (including an internal reconnect). Time spent waiting for other transactions (in

NICE

state Waiting for session) is not excluded. In case of [EXECUTE SCRIPT](#) the QUERY_TIMEOUT is applied to the script in whole, when reaching the timeout the script is terminated (including any statements being executed by the script). Please note that any changes of the QUERY_TIMEOUT within a script will only be applied when the script exits. The default value for QUERY_TIMEOUT is '0' (no restrictions). If the parameter NICE is set to 'ON', the session's priority will be reduced. The resource manager then divides the weight of the user by the number of currently active sessions. Hence, the session conserves resources, even across priority groups. Details about priorities can be found in [Section 3.3, “Priorities”](#).

CONSTRAINT_STATE_DEFAULT

This parameter defines the default state of constraints ('ENABLE' or 'DISABLE') in case the state wasn't explicitly specified during the creation (see also [CREATE TABLE](#) and [ALTER TABLE \(constraints\)](#)).

PROFILE

Activates/deactivates the profiling (values 'ON' or 'OFF'). For details see also [Section 3.9, “Profiling”](#).

SCRIPT_LANGUAGES

Defines the script language aliases. For details see [Section 3.6.5, “Expanding script languages using BucketFS”](#).

SQL_PREPROCESSOR_SCRIPT

Defines a preprocessor script. If such a script is specified (a regular script which was created via [CREATE SCRIPT](#)), then every executed SQL statement is preprocessed by that script. Please refer to [Section 3.8, “SQL Preprocessor”](#) for more information on SQL preprocessing. Details about the script language can be found in [Section 3.5, “Scripting”](#).



Please note that appropriate user privileges must exist for executing this script.



You can deactivate the preprocessing by specifying the empty string '' or NULL.

Example(s)

```
ALTER SESSION SET TIME_ZONE='EUROPE/BERLIN';
ALTER SESSION SET QUERY_TIMEOUT=120;
ALTER SESSION SET NLS_DATE_FORMAT='DDD-YYYY';
ALTER SESSION SET NLS_DATE_LANGUAGE='ENG';
SELECT TO_CHAR(TO_DATE('365-2007'), 'DAY-DD-MONTH-YYYY') TO_CHAR1;

TO_CHAR1
-----
MONDAY -31-DECEMBER -2007

ALTER SESSION SET NLS_NUMERIC_CHARACTERS=',.';
SELECT TO_CHAR(123123123.45, '999G999G999D99') TO_CHAR2;

TO_CHAR2
-----
123.123.123,45
```

ALTER SYSTEM

Purpose

System-wide parameters can be configured with this statement.

.

Prerequisite(s)

- The ALTER SYSTEM system privilege.

Syntax

alter_system::=



Note(s)

- The session-based parameters are initialized with the system-wide parameters (ALTER SYSTEM), but can be overwritten with the [ALTER SESSION](#) statement. The current settings can be found in the [EXA_PARAMETERS](#) system table.
- If a value is changed via ALTER SYSTEM, it will only impact new connections to the database.
- The following parameters can be set:

TIME_ZONE

Defines the time zone in which the values of type TIMESTAMP WITH LOCAL TIME ZONE are interpreted. Further information can be found in section [Date/Time data types](#) in [Section 2.3, “Data types”](#). The list of supported timezones can be found in the system table [EXA_TIME_ZONES](#). The function [SESSIONTIMEZONE](#) returns the current session time zone.

TIME_ZONE_BEHAVIOR

Defines the course of action for ambiguous and invalid timestamps within a certain time zone. Further information can be found in section [Date/Time data types](#) in [Section 2.3, “Data types”](#).

TIMESTAMP_ARITHMETIC_BEHAVIOR

Defines the behavior for +/- operators:

- INTERVAL - The difference of two datetime values is an interval, and when adding a decimal value to a timestamp, the number is rounded to an integer, so actually a certain number of full days is added.
- DOUBLE - The difference of two datetime values is a double, and when adding a decimal value to a timestamp, the fraction of days is added (hours, minutes, ...).

NLS_DATE_FORMAT

Sets the date format used for conversions between dates and strings. Possible formats are described in [Section 2.6.1, “Date/Time format models”](#).

NLS_TIMESTAMP_FORMAT

Sets the timestamp format used for conversions between timestamps and strings. Possible formats are described in [Section 2.6.1, “Date/Time format models”](#).

NLS_DATE_LANGUAGE

Sets the language of the date format used in abbreviated month and day formats and those written in full (see [Section 2.6.1, “Date/Time format models”](#)). Possible languages are English (ENG = Default) and German (DEU). The English language can be set using ENG or ENGLISH and the German language with DEU, DEUTSCH, and GERMAN.

NLS_FIRST_DAY_OF_WEEK

Defines the first day of a week (integer 1-7 for Monday-Sunday).

NLS_NUMERIC_CHARACTERS

Defines the decimal and group characters used for representing numbers. This parameter is also relevant to the use of numeric format models (see also [Section 2.6.2, “Numeric format models”](#)).

DEFAULT_LIKE_ESCAPE_CHARACTER

Defines the escape character for the LIKE predicate (see [Section 2.8, “Predicates”](#)) in case no explicit one was specified.

QUERY_CACHE

The parameter QUERY_CACHE defines the usage of a read cache for SELECT queries. If the syntactically identical query is sent multiple times (except upper/lower case, spaces, ...), then the database can read the result directly out of a cache instead of executing the query. This is

only applicable if the corresponding schema objects haven't changed in the meantime. The following values can be set:

- ON - The query cache is actively used, i.e. query results are read from and written into the cache.
- OFF - The query cache is not used.
- READONLY - Results are read from the cache, but additional new queries will not be cached.

Whether a query was returned from the cache can be determined by the column `EXECUTION_MODE` in the corresponding system tables (e.g. `EXA_SQL_LAST_DAY`).

QUERY_TIMEOUT

Defines how many seconds a statement may run before it is automatically aborted. When this point is reached, the statement may finish with an exception within a few seconds through an internal cancellation point. If this fails - for example because there are no such cancellation points (e.g. in LUA) or the query is slowed down due to disk operations - the query is terminated forcefully and the transaction is rolled back (including an internal reconnect). Time spent waiting for other transactions (in state `Waiting for session`) is not excluded. In case of [EXECUTE SCRIPT](#) the `QUERY_TIMEOUT` is applied to the script in whole, when reaching the timeout the script is terminated (including any statements being executed by the script). Please note that any changes of the `QUERY_TIMEOUT` within a script will only be applied when the script exits. The default value for `QUERY_TIMEOUT` is '0' (no restrictions).

CONSTRAINT_STATE_DEFAULT

This parameter defines the default state of constraints ('ENABLE' or 'DISABLE') in case the state wasn't explicitly specified during the creation (see also [CREATE TABLE](#) and [ALTER TABLE \(constraints\)](#)).

PROFILE

Activates/deactivates the profiling (values 'ON' or 'OFF'). For details see also [Section 3.9, “Profiling”](#).

SCRIPT_LANGUAGES

Defines the script language aliases. For details see [Section 3.6.5, “Expanding script languages using BucketFS”](#).

SQL_PREPROCESSOR_SCRIPT

Defines a preprocessor script. If such a script is specified (a regular script which was created via [CREATE SCRIPT](#)), then every executed SQL statement is preprocessed by that script. Please refer to [Section 3.8, “SQL Preprocessor”](#) for more information on SQL preprocessing. Details about the script language can be found in [Section 3.5, “Scripting”](#).



Please note that appropriate user privileges must exist for executing this script.



You can deactivate the preprocessing by specifying the empty string '' or NULL.

Example(s)

```
ALTER SYSTEM SET TIME_ZONE='EUROPE/BERLIN';
ALTER SYSTEM SET TIME_ZONE_BEHAVIOR='INVALID SHIFT AMBIGUOUS ST';
ALTER SYSTEM SET NLS_DATE_FORMAT='DAY-DD-MM-MONTH-YYYY';
ALTER SYSTEM SET NLS_DATE_LANGUAGE='DEU';
ALTER SYSTEM SET NLS_FIRST_DAY_OF_WEEK=1;
ALTER SYSTEM SET NLS_NUMERIC_CHARACTERS=', .';
ALTER SYSTEM SET QUERY_TIMEOUT=120;
ALTER SYSTEM SET CONSTRAINT_STATE_DEFAULT='DISABLE';
ALTER SYSTEM SET SQL_PREPROCESSOR_SCRIPT=my_schema.my_script;
```

OPEN SCHEMA

Purpose

A schema can be opened with this statement which affects the name resolution.

Prerequisite(s)

- None.

Syntax

open_schema ::=



Note(s)

- The currently opened schema can be identified by means of [CURRENT_SCHEMA](#).
- The schema can be exited with the [CLOSE SCHEMA](#) statement.
- If there is no schema opened, all the schema objects must be referenced using schema-qualified names (see also [Section 2.1.2, “SQL identifier”](#)).

Example(s)

```
OPEN SCHEMA my_schema;
```

CLOSE SCHEMA

Purpose

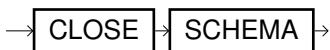
This statement is used to close the current schema which affects the name resolution.

Prerequisite(s)

- None.

Syntax

close_schema ::=



Note(s)

- If there is no schema opened, all the schema objects must be referenced using schema-qualified names (see also [Section 2.1.2, “SQL identifier”](#)).

Example(s)

```
OPEN SCHEMA my_schema;
SELECT * FROM my_table;
CLOSE SCHEMA;
SELECT * FROM my_table; -- Error: object MY_TABLE not found
SELECT * FROM my_schema.my_table;
```

DESC[RIBE]

Purpose

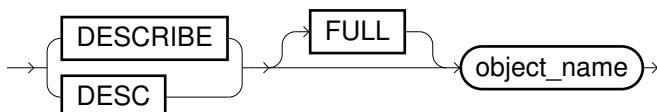
This statement is used to print column information(s) for a given table or view.

Prerequisite(s)

- If the object to be described is a table, one of the following conditions must be fulfilled:
 - The current user has one of the following system privileges: SELECT ANY TABLE (or SELECT ANY DICTIONARY in context of system tables, respectively), INSERT ANY TABLE, UPDATE ANY TABLE, DELETE ANY TABLE, ALTER ANY TABLE or DROP ANY TABLE.
 - The current user has any object privilege on the table.
 - The table belongs to the current user or one of his roles.
- If the object to be described is a view, one of the following conditions must be fulfilled:
 - The current user has one of the following system privileges: SELECT ANY TABLE or DROP ANY VIEW.
 - The current user has any object privilege on the view.
 - The view belongs to the current user or one of his roles.

Syntax

describe::=



Note(s)

- The SQL_TYPE column displays the datatype. In case of a string type, the used character set will be additionally shown (ASCII or UTF8).
- The NULLABLE column indicates whether the column is permitted to contain NULL values.
- The value of DISTRIBUTION_KEY shows whether the column is part of the distribution key (see also the [ALTER TABLE \(distribution\)](#) statement). For Views this value is always NULL.
- If you specify the option FULL, then the additional column COLUMN_COMMENT displays the column comment (cut to maximal 200 characters), if this was set either implicitly by the [CREATE TABLE](#) command or explicitly via the statement [COMMENT](#).
- DESCRIBE can be abbreviated by DESC (e.g., **DESC my_table;**).

Example(s)

```
CREATE TABLE t (i DECIMAL COMMENT IS 'id column',
                d DECIMAL(20,5),
                j DATE,
                k VARCHAR(5),
```

```

        DISTRIBUTED BY i;
DESCRIBE FULL t;

COLUMN_NAME SQL_TYPE           NULLABLE DISTRIBUTION_KEY COLUMN_COMMENT
----- ----- ----- -----
I      DECIMAL(18,0)    TRUE    TRUE          id column
D      DECIMAL(20,5)    TRUE    FALSE
J      DATE            TRUE    FALSE
K      VARCHAR(5)  UTF8   TRUE    FALSE

```

EXPLAIN VIRTUAL

Purpose

This statement is useful to analyze what an adapter script is pushing down to the underlying data source of a virtual object.

Prerequisite(s)

- Similar privileges as if you would just execute the contained query.

Syntax

explain_virtual::=

→ **EXPLAIN** → **VIRTUAL** → **(subquery)**

Note(s)

- Details about adapter scripts and virtual schemas can be found in [Section 3.7, “Virtual schemas”](#).
- If you access virtual objects in a query and use the EXPLAIN command, the actual query is not executed and no underlying data is transferred. The query is only compiled and optimized for pushing down as much logic as possible into the query for the underlying system. The result of that statement is then a table containing the effective queries for the external systems.
- You can use the EXPLAIN command as a subselect and thus process its result via SQL.
- Similar information about the details of a pushdown can also be found in profiling information, but this way it is far easier to access.

Example(s)

```

SELECT pushdown_id, pushdown_involved_tables, pushdown_sql FROM
(EXPLAIN VIRTUAL SELECT * FROM vs_impala.sample_07 WHERE total_emp>10000);

PUSHDOWN_ID PUSHDOWN_INV... PUSHDOWN_SQL
-----
1 SAMPLE_07          IMPORT FROM JDBC AT 'jdbc:impala:<shortened>'
                           STATEMENT 'SELECT * FROM `default`.`SAMPLE_07`'
                           WHERE 10000 < `TOTAL_EMP`'

```

RECOMPRESS

Purpose

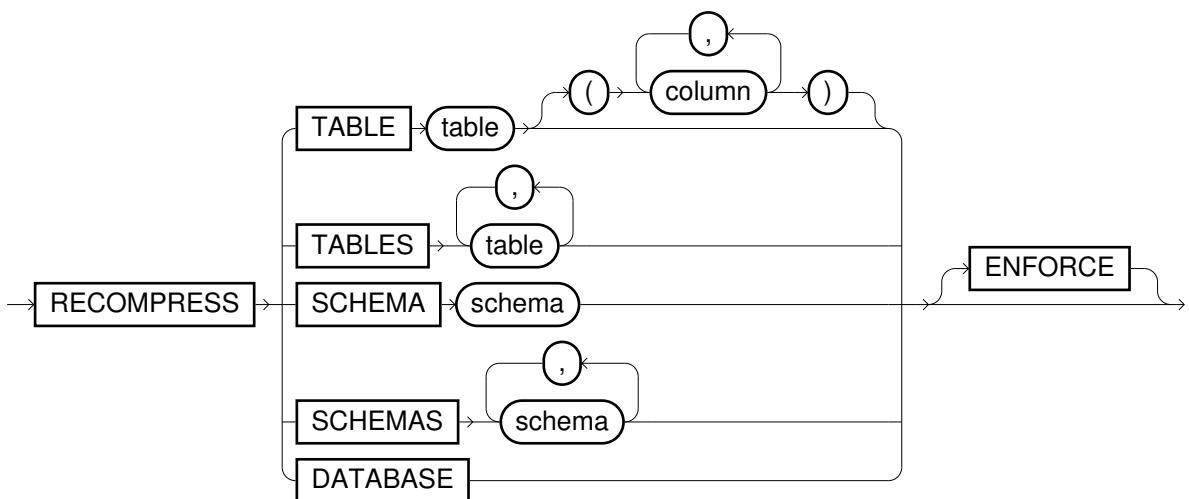
By using this statement you can improve the compression of tables.

Prerequisite(s)

- Access to all tables by the ownership of the user or any of its roles or by any of the modifying ANY TABLE system privileges or any modifying object privilege. Modifying privileges are all except SELECT.

Syntax

recompress::=



Note(s)

This command implies a **COMMIT** before and after recompressing any of the specified tables except if you use the single table alternative **RECOMPRESS TABLE!**

- The execution time of this command can take some time. A smart logic tries to only recompress those columns where a significant improvement can be achieved, but you can enforce the recompression using the ENFORCE option.
 - A recompression of a table especially makes sense after inserting a big amount of new data. But please note that afterwards, the compression ratios do not need to be significantly better than before.
 - You can also specify certain columns to only partially compress a table.
 - The compressed and raw data size of a table can be found in the system table [EXA_ALL_OBJECT_SIZES](#).

Example(s)

```
RECOMPRESS TABLE t1 (column_1);  
RECOMPRESS TABLES t2,t3 ENFORCE;
```

REORGANIZE

Purpose

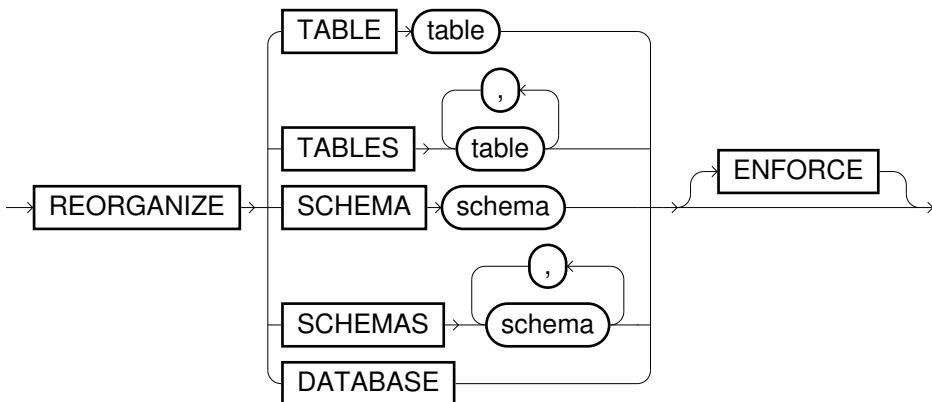
By using this command the database can be reorganized internally, which is necessary if the database cluster was enlarged. This statement redistributes the data across the nodes and reconstitutes the distribution status (see also [ALTER TABLE \(distribution\)](#)). That is why it is recommended that this statement is executed immediately after the cluster enlargement. Elsewise the system performance can even be worse than before the enlargement.

Prerequisite(s)

- Access to all tables by the ownership of the user or any of its roles or by any of the modifying ANY TABLE system privileges or any modifying object privilege. Modifying privileges are all except SELECT.

Syntax

reorganize ::=



Note(s)

- If you specify the option DATABASE, all existing tables are reorganized.
- Multiple tables are reorganized step by step and an implicit COMMIT is executed after each table. This leads to less transaction conflicts and improves the performance when accessing already reorganized tables as soon as possible.
- The reorganization consists of the following actions:
 - DELETE reorganization: refilling rows that are only marked as deleted (a kind of defragmentation). This happens automatically after reaching a certain threshold, but can be explicitly triggered with this command. See also [DELETE](#) statement for further details.
 - Recompressing the columns whose compression ratio decreased significantly over time.
 - Recreation of all internal indices.
 - TABLE redistribution: distributing the rows evenly across the cluster nodes and re-establishing the DISTRIBUTIVE BY status after a cluster enlargement.
- If you specify the ENFORCE option, all specified tables are reorganized. Otherwise only those tables are adjusted where this operation is absolutely necessary (e.g. due to a cluster enlargement or lots of rows marked as deleted).
- The execution time of this command can take some time. In particular each COMMIT potentially writes lots of data to discs.

Example(s)

```
REORGANIZE DATABASE;
```

TRUNCATE AUDIT LOGS

Purpose

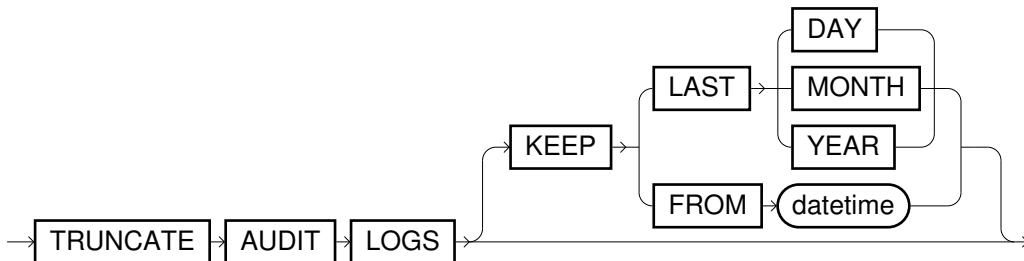
Via this command you can clear the data of the system tables [EXA_DB_AUDIT_SESSIONS](#), [EXA_DB_AUDIT_SQL](#), [EXA_USER_TRANSACTION_CONFLICTS_LAST_DAY](#) and [EXA_DB_TRANSACTION_CONFLICTS](#). This can be necessary e.g. if the gathered data volume is too big.

Prerequisite(s)

- The user requires the DBA role.

Syntax

truncate_audit_logs ::=



Note(s)

- By using the KEEP option you can keep the recent data.

Example(s)

```

TRUNCATE AUDIT LOGS;
TRUNCATE AUDIT LOGS KEEP LAST MONTH;
TRUNCATE AUDIT LOGS KEEP FROM '2009-01-01';
  
```

FLUSH STATISTICS

Purpose

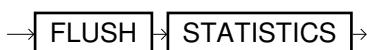
Statistical information in system tables is gathered continuously, but committed in certain periods. Using the FLUSH STATISTICS command, you can force this commit.

Prerequisite(s)

- None.

Syntax

flush_statistics ::=



Note(s)

- To show the newest flushed statistics, you may need to open a new transaction.
- This command generates additional load for the DBMS and should be used with caution. The statistical information is anyway updated every minute.
- The available statistical information is described in [Section A.2.3, “Statistical system tables”](#).

Example(s)

```
FLUSH STATISTICS;
COMMIT;
SELECT * FROM EXA_USAGE_LAST_DAY ORDER BY MEASURE_TIME DESC LIMIT 10;
```

PRELOAD

Purpose

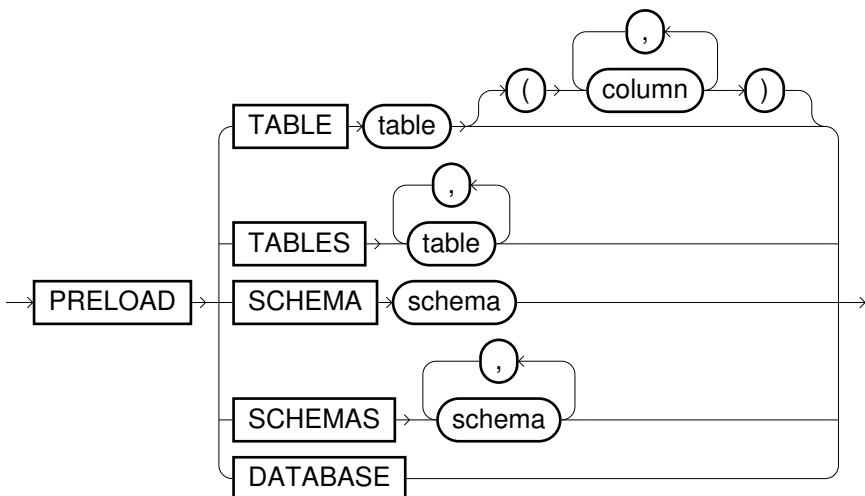
Loads certain tables or columns and the corresponding internal indices from disk in case they are not yet in the database cache. Due to Exasol's smart cache management we highly recommend to use this command only exceptionally. Otherwise you risk a worse overall system performance.

Prerequisite(s)

- Access to all tables by the ownership of the user or any of its roles or by any read/write system or object privileges.

Syntax

preload::=



Note(s)

- If you specify schemas or the complete database, the corresponding tables will be loaded into the cache.

Example(s)

```
PRELOAD TABLE t(i);
```

2.3. Data types

The data types and aliases supported by Exasol will be explained in this section. Beside the data types of the [ANSI SQL](#) standard some additional data types are implemented that ensure compatibility with other databases.

2.3.1. Overview of Exasol data types

The following table provides an overview of the defined data types in Exasol.

Table 2.2. Overview of Exasol data types

Exasol type (ANSI type)	Note
BOOLEAN	
CHAR(n)	$1 \leq n \leq 2,000$
DATE	
DECIMAL(p,s)	$s \leq p \leq 36$ $p \geq 1; s \geq 0$
DOUBLE PRECISION	
GEOMETRY[(srid)]	srid defines the coordinate system (see also EXA_SPATIAL_REF_SYS)
INTERVAL DAY [(p)] TO SECOND [(fp)]	$1 \leq p \leq 9, 0 \leq fp \leq 9$, accuracy precise to a millisecond
INTERVAL YEAR [(p)] TO MONTH	$1 \leq p \leq 9$
TIMESTAMP	Timestamp with accuracy precise to a millisecond
TIMESTAMP WITH LOCAL TIME ZONE	Timestamp which considers the session time zone
VARCHAR(n)	$1 \leq n \leq 2,000,000$

2.3.2. Data type details

Numeric data types

Exasol supports approximate and exact numerical data types. The difference when using the approximate data types arises from the possible rounding differences, which can occur due to the type of storage and the way computations are performed. Information losses of this nature do not occur when using exact numeric types because of the type of storage.

Exasol type (ANSI type)	Range (min., max.)	Precision	Note
<i>Exact numeric type</i>			
DECIMAL(p,s)	$\left[-\frac{10^p - 1}{10^s}, +\frac{10^p - 1}{10^s} \right]$	Precision, Scale: (p,s)	$s \leq p \leq 36$ $p \geq 1$ and $s \geq 0$
<i>Approximate numeric type</i>			
DOUBLE PRECISION	$[-1.7 \cdot 10^{308}, +1.7 \cdot 10^{308}]$	~15 decimal digits	About 15 digits can be stored, independent of the scale. The special element NaN is interpreted as NULL, the element Infinity is not supported.

Boolean data type

The [ANSI](#) SQL standard data type BOOLEAN is directly supported in Exasol.

Exasol type (ANSI type)	Range	Note
BOOLEAN	TRUE, FALSE, NULL/UNKNOWN	<p>Instead of boolean literals (see Range) the following numbers and strings can be interpreted:</p> <p>TRUE: 1, '1', 'T', 't', 'TRUE' (case-insensitive)</p> <p>FALSE: 0, '0', 'F', 'f', 'FALSE' (case-insensitive)</p>

Date/Time data types

Of the various data types of the [ANSI](#) SQL standard, Exasol currently supports the DATE and TIMESTAMP types. DATE corresponds to a date. In addition to the date, TIMESTAMP contains the time. Additionally, the data type TIMESTAMP WITH LOCAL TIME ZONE exists which considers the session time zone (see [ALTER SESSION](#) and [SESSIONTIMEZONE](#)) in calculations.

Exasol type (ANSI type)	Range	Note
DATE	01.01.0001 to 31.12.9999	
TIMESTAMP	01.01.0001 00:00:00.000 to 31.12.9999 23:59:59.999	The accuracy is limited to milliseconds!
TIMESTAMP WITH LOCAL TIME ZONE	01.01.0001 00:00:00.000 to 31.12.9999 23:59:59.999	Same range like the normal TIMESTAMP type, but this type considers the session time zone. The accuracy is limited to milliseconds!

Notes for data type TIMESTAMP WITH LOCAL TIME ZONE

- In case of TIMESTAMP WITH LOCAL TIME ZONE columns, the timestamps are internally normalized to UTC, while the input and output value is interpreted in the session time zone. Hence, users in different time zones can easily insert and display data without having to care about the internal storage. However, you should be aware that executing the similar SQL statement in sessions with different time zones can lead to different results.
- While TIMESTAMP is a simple structure consisting of year, month, day, hour, minute and second, data of type TIMESTAMP WITH LOCAL TIME ZONE represents a specific moment on the time axis. Internally, the data is normalized to UTC, because within the certain time zones exist time shifts (e.g. when switching from winter to summer time) and ambiguous periods (e.g. when switching from summer to winter time). If such problematic data is inserted within the local session time zone, the session value TIME_ZONE_BEHAVIOR (changeable by [ALTER SESSION](#)) defines the course of action.

The string for option TIME_ZONE_BEHAVIOR consists of two parts:

INVALID

When the time is shifted forward in a timezone, then a gap evolves. If timestamps are located within this gap, they can be treated in different ways:

SHIFT Corrects the value by adding the daylight saving time offset (typically one hour)

ADJUST Rounds the value to the first valid value after the time shift

NULIFY Sets the value to NULL

REJECT Throws an exception

AMBIGUOUS

When the time is shifted backward in a timezone, then ambiguous timestamps exist which can be treated in different ways:

ST	Interprets the value in Standard Time (ST)
DST	Interprets the value in Daylight Saving Time (DST)
NULLIFY	Sets the value to NULL
REJECT	Throws an exception

Thus, the following combinations can be defined:

'INVALID SHIFT AMBIGUOUS ST'
'INVALID SHIFT AMBIGUOUS DST'
'INVALID SHIFT AMBIGUOUS NULLIFY'
'INVALID SHIFT AMBIGUOUS REJECT'
'INVALID ADJUST AMBIGUOUS ST'
'INVALID ADJUST AMBIGUOUS DST'
'INVALID ADJUST AMBIGUOUS NULLIFY'
'INVALID ADJUST AMBIGUOUS REJECT'
'INVALID NULLIFY AMBIGUOUS ST'
'INVALID NULLIFY AMBIGUOUS DST'
'INVALID NULLIFY AMBIGUOUS NULLIFY'
'INVALID NULLIFY AMBIGUOUS REJECT'
'INVALID REJECT AMBIGUOUS ST'
'INVALID REJECT AMBIGUOUS DST'
'INVALID REJECT AMBIGUOUS NULLIFY'
'INVALID REJECT AMBIGUOUS REJECT'

- When casting between the data types `TIMESTAMP` and `TIMESTAMP WITH LOCAL TIME ZONE`, the session time zone is evaluated and the `TIMESTAMP WITH LOCAL TIME ZONE` transformed into a normal `TIMESTAMP`. This is the similar approach to displaying a `TIMESTAMP WITH LOCAL TIME ZONE` value in e.g. EXAplus, since then the internally UTC-normalized value is also converted into a normal `TIMESTAMP`, considering the session time zone.
- Special literals do not exist for the data type `TIMESTAMP WITH LOCAL TIME ZONE`. The normal `TIMESTAMP` literals are expected (see [Section 2.5, “Literals”](#)), and the corresponding moment on the time axis is defined via the session time zone. Details about the arithmetic on datetime values and the datetime functions can be found in the corresponding chapters ([Section 2.9.1, “Scalar functions”](#) and [Section 2.7, “Operators”](#)).

- Please mention that timestamp values logged in statistical system tables are interpreted in the database time zone ([DBTIMEZONE](#)) which can be set via EXAoperation. This is especially relevant if you want to use the different functions for the current timestamp:

<code>SYSTIMESTAMP</code>	Returns the current timestamp, interpreted in the database time zone (DB-TIMEZONE)
<code>CURRENT_TIMESTAMP</code>	Returns the current timestamp, interpreted the session time zone (SESSION-TIMEZONE)
<code>LOCALTIMESTAMP</code>	Synonym for <code>CURRENT_TIMESTAMP</code>
<code>NOW</code>	Synonym for <code>CURRENT_TIMESTAMP</code>

- The list of supported timezones can be found in the system table [EXA_TIME_ZONES](#).

Interval data types

Via the both interval data types you can define time periods which are especially useful for datetime arithmetic.

Exasol type (ANSI type)	Range	Note
INTERVAL YEAR [(p)] TO MONTH (e.g. '5-3')	'-999999999-11' to '999999999-11'	$1 \leq p \leq 9$ (default: 2)
INTERVAL DAY [(p)] TO SECOND [(fp)] (e.g. '2 12:50:10.123')	'-999999999 23:59:59.999' to '999999999 23:59:59.999'	$1 \leq p \leq 9$ (default: 2), $0 \leq fp \leq 9$ (default: 3) The accuracy is limited to milliseconds!

Geospatial data type

The geospatial data type can store various geometry objects. Details can be found in [Section 2.4, “Geospatial data”](#).

Exasol type (ANSI type)	Possible Elements	Note
GEOMETRY[(srid)]	POINT, LINESTRING, POLYGON, MULTIPOINT, MULTILINESTRING, MULTIPOLYGON, GEOMETRYCOLLECTION	srid defines the coordinate system (see also EXA_SPATIAL_REF_SYS)

String data types

In Exasol two [ANSI](#) SQL standard types are supported for characters: CHAR and VARCHAR.

The CHAR(n) data type has a fixed, pre-defined length n. When storing shorter strings, these are filled with spacing characters ("padding").

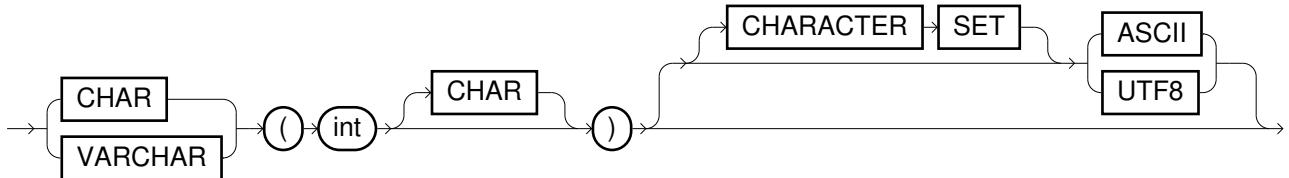
VARCHAR(n) can contain any string of the length n or smaller. These strings are stored in their respective length.

The length of both types is limited to 2,000 characters (CHAR) and 2,000,000 characters (VARCHAR), respectively both can be defined with a character set: ASCII or UTF8 (Unicode). If you omit this definition, UTF8 will be used.



An empty string is evaluated as NULL value.

stringtype_definition ::=



The character set of a certain column can be displayed by using the command [DESC\[RIBE\]](#).

Exasol type (ANSI type)	Character Set	Note
CHAR(n)	ASCII, UTF8	$1 \leq n \leq 2,000$
VARCHAR(n)	ASCII, UTF8	$1 \leq n \leq 2,000,000$

2.3.3. Data type aliases

For reasons of compatibility, several aliases are implemented in Exasol.

Table 2.3. Summary of Exasol aliases

Alias	Exasol type equivalent	Note
BIGINT	DECIMAL(36,0)	
BOOL	BOOLEAN	
CHAR	CHAR(1)	
CHAR VARYING(n)	VARCHAR(n)	$1 \leq n \leq 2,000,000$
CHARACTER	CHAR(1)	
CHARACTER LARGE OBJECT	VARCHAR(1)	
CHARACTER LARGE OBJECT(n)	VARCHAR(n)	$1 \leq n \leq 2,000,000$
CHARACTER VARYING(n)	VARCHAR(n)	$1 \leq n \leq 2,000,000$
CLOB	VARCHAR(2000000)	
CLOB(n)	VARCHAR(n)	$1 \leq n \leq 2,000,000$
DEC	DECIMAL(18,0)	
DEC(p)	DECIMAL(p,0)	$1 \leq p \leq 36$
DEC(p,s)	DECIMAL(p,s)	$s \leq p \leq 36$ $p \geq 1; s \geq 0$
DECIMAL	DECIMAL(18,0)	
DECIMAL(p)	DECIMAL(p,0)	$1 \leq p \leq 36$
DOUBLE	DOUBLE PRECISION	
FLOAT	DOUBLE PRECISION	
INT	DECIMAL(18,0)	
INTEGER	DECIMAL(18,0)	
LONG VARCHAR	VARCHAR(2000000)	
NCHAR(n)	CHAR(n)	
NUMBER	DOUBLE PRECISION	Possible loss in precision
NUMBER(p)	DECIMAL(p,0)	$1 \leq p \leq 36$
NUMBER(p,s)	DECIMAL(p,s)	$s \leq p \leq 36$ $p \geq 1; s \geq 0$
NUMERIC	DECIMAL(18,0)	
NUMERIC(p)	DECIMAL(p,0)	$1 \leq p \leq 36$
NUMERIC(p,s)	DECIMAL(p,s)	$s \leq p \leq 36$ $p \geq 1; s \geq 0$
NVARCHAR(n)	VARCHAR(n)	$1 \leq n \leq 2,000,000$
NVARCHAR2(n)	VARCHAR(n)	$1 \leq n \leq 2,000,000$
REAL	DOUBLE PRECISION	
SHORTINT	DECIMAL(9,0)	
SMALLINT	DECIMAL(9,0)	
TINYINT	DECIMAL(3,0)	
VARCHAR2(n)	VARCHAR(n)	$1 \leq n \leq 2,000,000$

2.3.4. Type conversion rules

In many places in SQL queries, certain data types are expected, e.g. the CHAR or VARCHAR data type in the string function [SUBSTR\[ING\]](#). Should a user wish to use a different type, it is recommendable to work with explicit conversion functions (see also the list of [Conversion functions](#)).

If explicit conversion functions are not used, the system attempts to perform an implicit conversion. If this is not possible or if one single value cannot be successfully converted during the computation, the system generates an error message.

The following table provides an overview of the permissible implicit conversions.

Table 2.4. Possible implicit conversions

Source \ Target data type	DECIMAL	DOUBLE	BOOLEAN	DATE / TIMESTAMP	INTERVAL YEAR [(p)] TO MONTH	INTERVAL DAY [(p)] TO SECOND [(fp)]	GEO-METRY	CHAR / VARCHAR
DECIMAL	✓	✓	✓	–	–	–	–	✓
DOUBLE	✓	✓	✓	–	–	–	–	✓
BOOLEAN	✓	✓	✓	–	–	–	–	✓
DATE / TIMESTAMP	–	–	–	✓	–	–	–	✓
INTERVAL YEAR [(p)] TO MONTH	–	–	–	–	✓	–	–	✓
INTERVAL DAY [(p)] TO SECOND [(fp)]	–	–	–	–	–	✓	–	✓
GEOMETRY	–	–	–	–	–	–	✓	✓
CHAR / VARCHAR	✓	✓	✓	✓	✓	✓	✓	✓

Symbol meaning:

- ✓ Implicit conversion always works
- ✗ Implicit conversion possible, but errors may occur if single values cannot be converted
- Implicit conversion not possible

Comments:

- When converting DECIMAL to DOUBLE rounding inaccuracies may occur.
- When converting DECIMAL or DOUBLE to BOOLEAN, 1 is transformed to TRUE, 0 to FALSE and NULL to NULL. It should be noted that other data type compatible representations of the values 1 and 0 can be used. For example, 1 = 1.00 (DECIMAL(3.2)) = 1.0 (DECIMAL(2.1)).
- When converting from BOOLEAN to DECIMAL or DOUBLE, TRUE is transformed to 1, FALSE to 0 and NULL to NULL.
- When converting from BOOLEAN to CHAR(n) or VARCHAR(n), TRUE is transformed to 'True', and FALSE to 'False'.
- The conversion of GEOMETRY objects with different coordinate systems is not supported.
- Conversions to CHAR(n) or VARCHAR(n) always function if n is large enough and all characters exist in the target character set.
- Conversions from CHAR(n) or VARCHAR(n) to another data type always succeed if the data to be converted conforms to the target data type.
- When converting from CHAR(n) or VARCHAR(n) to DATE or TIMESTAMP, the current format model must be taken into account (see also [Section 2.6.1, "Date/Time format models"](#)).

- When converting from CHAR(n) or VARCHAR(n) to BOOLEAN, you can use strings '0', 'F', 'f' or 'FALSE' (case-insensitive) for value FALSE and strings '1', 'T', 't' or 'TRUE' (case-insensitive) for value TRUE.
- In operations with multiple operands (e.g. the operators +,-,/,*) the operands are implicitly converted to the biggest occurring data type (e.g. DOUBLE is bigger than DECIMAL) before executing the operation. This rule is also called numeric precedence.

In the following example an implicit conversion is conducted in order to insert the BOOLEAN entry into the DECIMAL column of the created table.

Example(s)

```
CREATE TABLE t(d DECIMAL);

-- Implicit conversion from BOOLEAN to DECIMAL
INSERT INTO t VALUES TRUE, NULL, FALSE;

SELECT * FROM t;

D
-----
 1
 0
```

2.3.5. Default values

Introduction

Default values are preconfigured values which are always inserted into table columns instead of NULL if no explicit value is specified during an insert operation (e.g. [INSERT](#)).

Example

```
CREATE TABLE telephonelist(
    name          VARCHAR(10),
    phone_number VARCHAR(10),
    type          VARCHAR(10) DEFAULT 'home',
    alterationtime TIMESTAMP DEFAULT CURRENT_TIMESTAMP);

INSERT INTO telephonelist (name, phone_number) VALUES ('Meyer', '1234');
INSERT INTO telephonelist (name, phone_number) VALUES ('Müller', '5678');
INSERT INTO telephonelist (name, type, phone_number)
    VALUES ('Meyer', 'work', '9999');
UPDATE telephonelist SET name='Meier', alterationtime=DEFAULT
    WHERE phone_number='1234';

SELECT * FROM telephonelist;

NAME      PHONE_NUMBER TYPE      ALTERATIONTIME
-----  -----
Meier     1234          home     2010-12-13 16:39:02.393000
Müller   5678          home     2010-12-13 16:39:00.823000
Meyer     9999          work     2010-12-13 16:39:01.62600
```

Statements that use and manipulate default values

Default values can be created, altered and deleted with the following statements:

- [CREATE TABLE](#) (in the column definition)
- [ALTER TABLE](#) (column)
 - [ADD COLUMN](#) (in the column definition)
 - [MODIFY COLUMN](#) (in the column definition)
 - [ALTER COLUMN DEFAULT](#) with SET DEFAULT or DROP DEFAULT

Default values are explicitly or implicitly used in tables with the following statements:

- [INSERT](#): DEFAULT as a 'value' for a column (INSERT INTO t(i,j,k) VALUES (1, DEFAULT,5)) or DEFAULT VALUES for all columns (INSERT INTO t DEFAULT VALUES) or implicitly at column selection (for the columns that were not selected)
- [IMPORT](#): implicitly at column selection (for the columns that were not selected)
- [UPDATE](#): DEFAULT as a 'value' for a column (SET i=DEFAULT)
- [MERGE](#): DEFAULT as a 'value' for a column in the INSERT and UPDATE parts or implicitly in the INSERT part at column selection (for the columns that were not selected)
- [ADD COLUMN](#): if a default value was specified for the inserted column with DEFAULT

Permitted default values

The following expressions can be used as default values:

- Constants
- Constant values at evaluation time such as [CURRENT_USER](#) or [CURRENT_DATE](#)
- Value expressions, which only contain the functions of the two above mentioned expressions



Default values are limited to 2000 characters.

Examples

```
CREATE TABLE t (i DECIMAL, vc VARCHAR(100) );
ALTER TABLE t ALTER COLUMN i SET DEFAULT 1;
ALTER TABLE t ALTER COLUMN i SET DEFAULT 1+2;
ALTER TABLE t ALTER COLUMN vc SET DEFAULT 'abc'||TO_CHAR(CURRENT_DATE);
ALTER TABLE t ALTER COLUMN vc SET DEFAULT CURRENT_USER;
```

Display of default values

Default values can be displayed in the column COLUMN_DEFAULT of system tables [EXA_ALL_COLUMNS](#), [EXA_DBA_COLUMNS](#), and [EXA_USER_COLUMNS](#).

Example:

```
CREATE TABLE t (i DECIMAL DEFAULT 3+4);

SELECT column_default FROM exa_all_columns
    WHERE column_table='t'
        AND column_name='i';

COLUMN_DEFAULT
-----
3+4
```

Possible sources of error

In almost all cases, default-values should work as expected. However, a few things should be considered to ensure error-free operation:

- With **MODIFY COLUMN**, an existing default value is adopted when changing the data type. If the old default value is not appropriate for the new data type, an error message is given.
- If entries used as default values could have different lengths and potentially might not fit in the table column (e.g. **CURRENT_USER** or **CURRENT_SCHEMA**), an error message is given at the time of insertion if the value still fit when the default value was set, but does not fit any more. In this case, the insertion will not be performed. If in doubt, the default value should be reduced to the length of the column via **SUBSTR[ING]**.
- Interpretation of default values can depend on the format model (e.g. a DATE value). In this case it is advisable to explicitly enforce the format (e.g. by means of **TO_DATE** with format specification).

Example

```
CREATE TABLE t1 (d DATE DEFAULT '2006-12-31');
CREATE TABLE t2 (d DATE DEFAULT TO_DATE('2006-12-31', 'YYYY-MM-DD'));
ALTER SESSION SET NLS_DATE_FORMAT='DD.MM.YYYY';

-- with INSERT in t1 an error message occurs but not with t2
INSERT INTO t1 DEFAULT VALUES;
INSERT INTO t2 DEFAULT VALUES;
```

2.3.6. Identity columns

Introduction

By the use of identity columns you can generate ids. They are similar to default values, but evaluated dynamically.

Example

```
CREATE TABLE actors (id      INTEGER IDENTITY,
                     lastname VARCHAR(20),
                     surname VARCHAR(20));

INSERT INTO actors (lastname, surname) VALUES
  ('Pacino', 'Al'),
  ('Willis', 'Bruce'),
  ('Pitt', 'Brad');

SELECT * FROM actors;

ID      LASTNAME      SURNAME
-----  -----
1      Pacino        Al
2      Willis        Bruce
3      Pitt          Brad
```

Notes

- The content of an identity column applies to the following rules:

- If you specify an explicit value for the identity column while inserting a row, then this value is inserted.
- In all other cases monotonically increasing numbers are generated by the system, but gaps can occur between the numbers.
- The current value of the number generator can be changed via [ALTER TABLE \(column\)](#). Explicit inserted values have no influence on the number generator.
- Via DML statements ([INSERT](#), [IMPORT](#), [UPDATE](#), [MERGE](#)) you can anytime manipulate the values of an identity column.



You should not mistake an identity column with a constraint, i.e. identity columns do not guarantee unique values. But the values are unique as long as values are inserted only implicitly and are not changed manually.

- Identity columns must have an exact numeric data type without any scale (`INTEGER`, `DECIMAL(x , 0)`). The range for the generated values is limited by the data type.
- Tables can have at most one identity column.
- A column cannot be an identity column and have a default value at the same time.

Statements to set and change identity columns

Identity columns can be created, changed and dropped:

- [CREATE TABLE](#) (within the column definition)
- [ALTER TABLE \(column\)](#)
 - [ADD COLUMN](#) (within the column definition)
 - [MODIFY COLUMN](#) (within the column definition)
 - [ALTER COLUMN IDENTITY](#) using `SET IDENTITY` or `DROP IDENTITY`

The dynamically created values of an identity column are explicitly or implicitly used in the following statements:

- [INSERT](#): DEFAULT as 'value' for a column (`INSERT INTO t(i,j,k) VALUES (1,DEFAULT,5)`) or implicitly in case of a column selection where the identity column is not included
- [IMPORT](#): implicitly in case of a column selection where the identity column is not included
- [UPDATE](#): DEFAULT as 'value' for a column (`SET i=DEFAULT`)
- [MERGE](#): DEFAULT as 'value' for a column within the `INSERT` and `UPDATE` parts or implicitly in the `INSERT` part in case of a column selection where the identity column is not included
- [ADD COLUMN](#): if the added column is an identity column

Display of identity columns

The specified identity columns can be listed via the column `COLUMN_IDENTITY` of the system tables [EXA_ALL_COLUMNS](#), [EXA_DBA_COLUMNS](#), and [EXA_USER_COLUMNS](#). The entry of identity columns is the last generated number.

Example:

```
CREATE TABLE t (id INTEGER IDENTITY(30));
SELECT column_identity FROM exa_all_columns
    WHERE column_table='T'
        AND column_name='ID';

COLUMN_IDENTITY
-----
```

2.4. Geospatial data

By the use of geospatial data you can store and analyze geographical information. Points, LineStrings and areas are defined through coordinates and saved in Exasol as GEOMETRY objects. GEOMETRY columns can optionally have a SRID (a reference coordinate system, see also [EXA_SPATIAL_REF_SYS](#)) which is a kind of constraint.

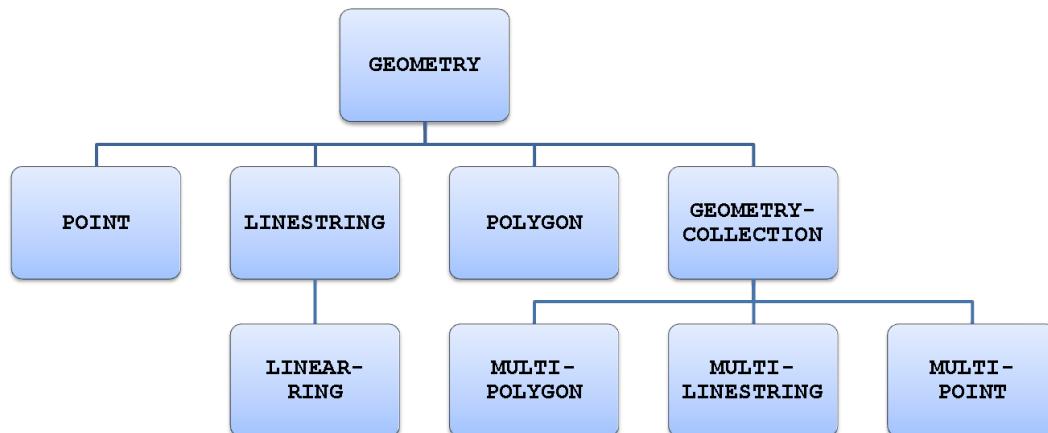
 Please note that the geospatial data feature is part of the Advanced Edition of Exasol.

 Please note that GEOMETRY columns are filled with strings (like e.g. 'POINT(2 5)'). If you read this data externally by the drivers, this data is automatically converted to strings. The same applies for the commands **IMPORT** and **EXPORT**.

For geospatial objects, a multitude of functions are provided to execute calculations and operations.

2.4.1. Geospatial objects

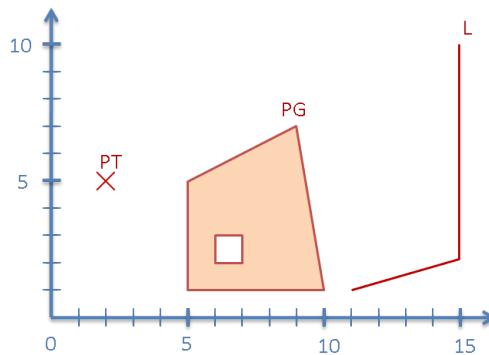
In the following table you find the different types of geospatial objects in Exasol.



Constructor	Description
Geometry	General abstraction of any geospatial objects
POINT(x y)	Point within the two-dimensional area
LINESTRING(x y, x y, ...)	LineString which connects several two-dimensional Points
LINEARRING(x y, x y, ...)	A linear ring is a LineString whose start and end points are identical.
POLYGON((x y, ...), [(x y, ...), ...])	Area which is defined by a linear ring and an optional list of holes within this area
GEOMETRYCOLLECTION(geometry, ...)	A collection of any geospatial objects
MULTIPOINT(x y, ...)	Set of Points
MULTILINESTRING((x y, ...), ...)	Set of LineStrings
MULTIPOLYGON((x y, ...), ...)	Set of Polygons

 Instead of numerical arguments you can also use the keyword **EMPTY** for creating the empty set of an object (e.g. POLYGON EMPTY)

Examples



```

POINT(2 5)                                -- PT
LINESTRING(10 1, 15 2, 15 10)            -- L
POLYGON((5 1, 5 5, 9 7, 10 1, 5 1),
        (6 2, 6 3, 7 3, 7 2, 6 2))       -- PG

MULTIPOINT(0.1 1.4, 2.2 3, 1 6.4)
MULTILINESTRING((0 1, 2 3, 1 6), (4 4, 5 5))
MULTIPOLYGON(((0 0, 0 2, 2 2, 3 1, 0 0)),
             ((4 6, 8 9, 12 5, 4 6), (8 6, 9 6, 9 7, 8 7, 8 6)))

GEOMETRYCOLLECTION(POINT(2 5), LINESTRING(1 1, 15 2, 15 10))

```

2.4.2. Geospatial functions

Exasol provides many functions to execute operations and mathematical calculations on geospatial objects. The arguments of the functions mainly consist of geospatial objects as described above (*p*=*POINT*, *ls*=*LINESTRING*, *mls*=*MULTILINESTRING*, *pg*=*POLYGON*, *mp*=*MULTIPOLYGON*, *g*=*GEOMETRY*, *gc*=*GEOMETRYCOLLECTION*, *n*=*Integer*).

Function	Description
Point Functions	
ST_X(<i>p</i>)	x coordinate of a Point.
ST_Y(<i>p</i>)	y coordinate of a Point.
(Multi-)LineString Functions	
ST_ENDPOINT(<i>ls</i>)	End point of a LineString.
ST_ISCLOSED(<i>mls</i>)	Defines whether all contained LineStrings are rings, i.e. whether their start and end points are identical.
ST_ISRING(<i>ls</i>)	Defines whether a LineString is a closed ring, i.e. start and end points are identical.
ST_LENGTH(<i>mls</i>)	Length of a LineString or the sum of lengths of all objects of a MultiLineString.
ST_NUMPOINTS(<i>ls</i>)	Number of Points within the LineString.
ST_POINTN(<i>ls</i> , <i>n</i>)	The <i>n</i> -th point of a LineString, starting with 1. Returns NULL if ST_NUMPOINTS(<i>ls</i>)< <i>n</i> .
ST_STARTPOINT(<i>ls</i>)	Start point of a LineString.
(Multi-)Polygon Functions	
ST_AREA(<i>mp</i>)	Area of a polygon or sum of areas of all objects of a MultiPolygon.

Function	Description				
ST_EXTERIORRING(pg)	Outer ring of the object.				
ST_INTERIORRINGN(pg, n)	The n-th hole of a Polygon, starting with 1. Returns NULL if ST_NUMINTERIORRINGS(pg)<n.				
ST_NUMINTERIORRINGS(pg)	Number of holes within a Polygon.				
GeometryCollection Functions					
ST_GEOMETRYN(gc, n)	The n-th object of a GeometryCollection, starting with 1. Returns NULL if ST_NUMGEOMETRIES(gc)<n.				
ST_NUMGEOMETRIES(gc)	Number of objects within a collection of geometry objects.				
General Functions					
ST_BOUNDARY(g)	Geometric boundary of a geospatial object (e.g. the end points of a LineString or the outer LinearRing of a Polygon).				
ST_BUFFER(g, n)	Returns a geospatial object, whose points have maximal distance n to the first argument. This is similar to a kind of extension of the borders of an object. Around edges some kind of divided circle is created which is approximated by a number of points.				
ST_CENTROID(g)	Geometric center of mass of an object.				
ST_CONTAINS(g, g)	Defines whether the first object fully contains the second one.				
ST_CONVEXHULL(g)	Convex hull of a geospatial object.				
ST_CROSSES(g, g)	Defines whether the two objects <i>cross</i> each other. This is the case if <ul style="list-style-type: none"> • The intersection is not empty and does not equal to one of the objects • The dimension of the intersection is smaller than the maximal dimension of both arguments 				
ST_DIFFERENCE(g, g)	Difference set of two geospatial objects.				
ST_DIMENSION(g)	Dimension of a geospatial object (e.g. 0 for Points, 1 for LineStrings and 2 for Polygons).				
ST_DISJOINT(g, g)	Defines whether two geospatial objects are disjoint, i.e. their intersection is empty.				
ST_DISTANCE(g, g)	Minimal distance between two geospatial objects.				
ST_ENVELOPE(g)	Smallest rectangle which contains the geospatial object.				
ST_EQUALS(g, g)	Defines whether two geospatial objects describe the same geometric object. For two objects g1 and g2 this is the case if ST_WITHIN(g1,g2) and ST_WITHIN(g2,g1).				
ST_FORCE2D(g)	Makes a 2D object out of a 3D object, meaning that it ignores the third coordinate. This function can be necessary since Exasol does not support 3D objects.				
ST_GEOGRAPHYTYPE(g)	Type of the geospatial object as string (e.g. 'POINT' or 'MULTIPOLYGON').				
ST_INTERSECTION(g, g)	Intersection of two geospatial objects. This function can also be used as aggregate function.				
ST_INTERSECTS(g, g)	Defines whether an intersection of two geospatial objects exists.				
ST_ISEMPTY(g)	Defines whether the object is the empty set.				
ST_ISSIMPLE(g)	Defines whether a geospatial objects is <i>simple</i> : <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 30%;">POINT</td> <td>Always simple.</td> </tr> <tr> <td>MULTIPOINT</td> <td>Simple if no points are identical.</td> </tr> </table>	POINT	Always simple.	MULTIPOINT	Simple if no points are identical.
POINT	Always simple.				
MULTIPOINT	Simple if no points are identical.				

Function	Description	
	LINESTRING	Simple if does not pass through the same Point twice (except start and end points).
	MULTILINESTRING	Simple if all LineStrings are simple and the intersections are only in the boundaries of both objects.
	POLYGON	Always simple.
	MULTIPOLYGON	Always simple.
ST_OVERLAPS(g,g)	Defines whether two geospatial objects overlap. This is the case if the objects are not identical, their intersection is not empty and has the same dimension as the two objects.	
ST_SETSRID(g,srid)	Sets the SRID for a geometry object (the coordinate system, see also EXA_SPATIAL_REF_SYS).	
ST_SYMDIFFERENCE(g,g)	Symmetric difference set of two geospatial objects.	
ST_TOUCHES(g,g)	Defines whether two geospatial objects <i>touch</i> each other. This is the case if the intersection is not empty and is only located on the boundaries of the objects (see ST_BOUNDARY).	
ST_TRANSFORM(g,srid)	Converts a geospatial object into the given reference coordinate system (see also EXA_SPATIAL_REF_SYS).	
ST_UNION(g,g)	Union set of two geospatial objects. This function can also be used as aggregate function.	
ST_WITHIN(g,g)	Defines whether the first object is fully contained by the second one (opposite of ST_CONTAINS).	

Examples

```

SELECT ST_Y('POINT (1 2)');
--> 2

SELECT ST_ENDPOINT('LINESTRING (0 0, 0 1, 1 1)');
--> POINT (1 1)

SELECT ST_ISRING('LINESTRING (0 0, 0 1, 1 1, 0 0)');
--> TRUE

SELECT ST_LENGTH('LINESTRING (0 0, 0 1, 1 1)');
--> 2

SELECT ST_BOUNDARY('LINESTRING (0 0, 1 1, 2 2)');
--> MULTIPOLYGON ((0 0, 2 2, 2 2, 1 1, 0 0))

SELECT ST_AREA('POLYGON ((0 0, 0 4, 4 4, 4 0, 0 0),
                    (1 1, 1 2, 2 2, 2 1, 1 1))');
--> 15 (=16-1)

SELECT ST_DISTANCE('POLYGON ((0 0, 0 4, 4 4, 4 0, 0 0))',
                  'POINT(12 10)');
--> 10

```

2.5. Literals

Literals represent constants, which possess a specific value as well as a corresponding data type. Literals can be entered into tables as values or used within queries as constants, e.g. for comparisons or function parameters. If the literal displays a different type to the assigned or comparison type, the "smaller" data type is implicitly converted.

Literals are grouped as follows: numeric literals, Boolean literals, date/time literals and string literals. There is also a special literal for the NULL value.

Examples

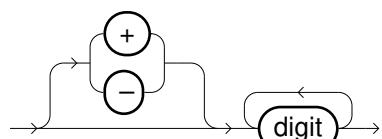
123	Integer number (integral decimal number)
-123.456	Decimal number
1.2345E-15	Double value
TRUE	Boolean value
DATE '2007-03-31'	Date
TIMESTAMP '2007-03-31 12:59:30.123'	Timestamp
INTERVAL '13-03' YEAR TO MONTH	Interval (YEAR TO MONTH)
INTERVAL '1 12:00:30.123' DAY TO SECOND	Interval (DAY TO SECOND)
'ABC'	String
NULL	NULL value

2.5.1. Numeric literals

Numeric literals represent a number. Distinction is made between exact numeric values and floating point numbers.

The following numeric literals are distinguished between:

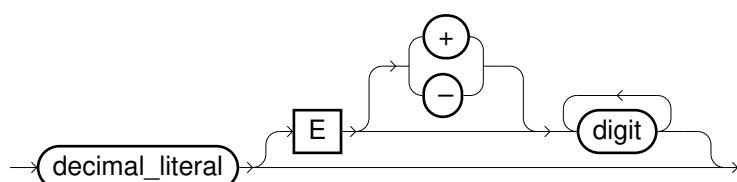
integer_literal::=



decimal_literal::=



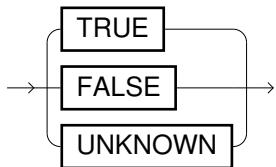
double_literal::=



2.5.2. Boolean literals

There are only three valid values for Boolean literals: TRUE, FALSE and UNKNOWN (which is similar to NULL). Instead of a literal you can also use strings (e.g. 'T' or 'True') or numbers (0 and 1). For details see [Boolean data type](#).

boolean_literal ::=



2.5.3. Date/Time literals

Date/Time literals represent a particular point in time. They have a fixed format, therefore, they are not as flexible as the [TO_DATE](#) or [TO_TIMESTAMP](#) conversion functions. In most cases, these functions should be given priority over literals (for more information see [Section 2.9, “Built-in functions”](#) as well as [Section 2.6.1, “Date/Time format models”](#)).

date_literal ::=



timestamp_literal ::=



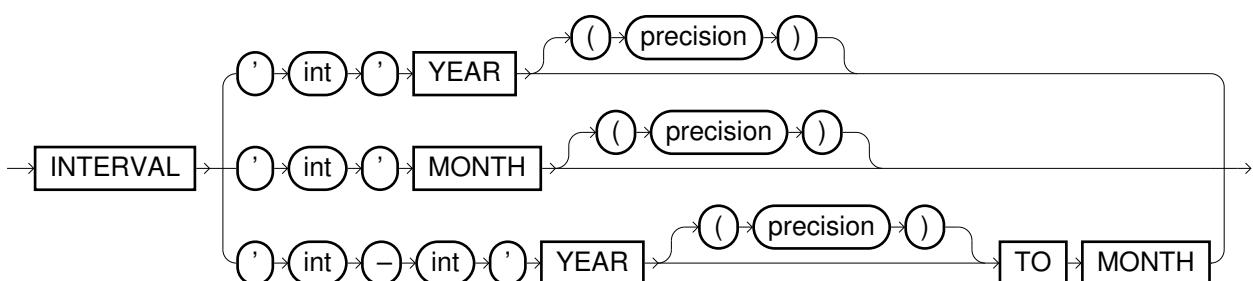
The following formats are specified for string:

Type	Format	Explanation
DATE	'YYYY-MM-DD'	Fixed format for DATE literals ('Year-Month-Day')
TIMESTAMP	'YYYY-MM-DD HH:MI:SS.FF3'	Fixed format for TIMESTAMP literals ('Year-Month-Day Hours-Minutes-Seconds.Milliseconds')

2.5.4. Interval literals

Interval literals describe an interval of time and are very useful for datetime arithmetic.

interval_year_to_month_literal ::=

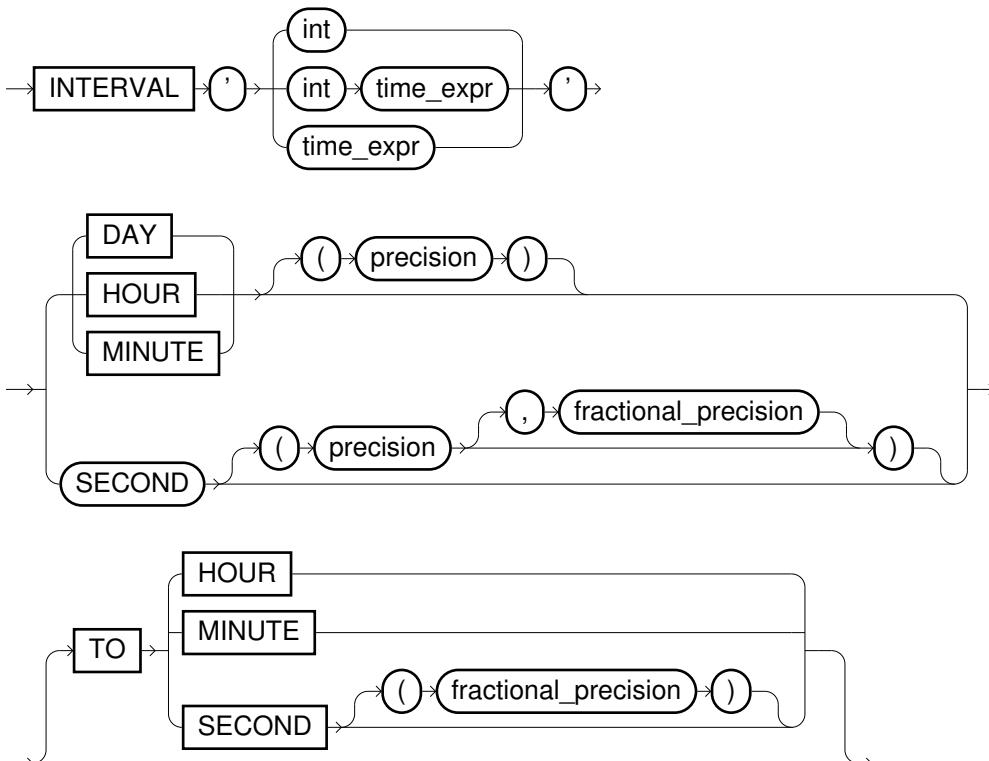


Notes:

- int[-int] defines integer values for the number of years and months. In case of YEAR TO MONTH you have to specify int-int.
- The optional parameter precision (1-9) specifies the maximal number of digits. Without this parameter 2 digits are allowed (months from 0 to 11).
- Examples:

INTERVAL '5' MONTH	Five months
INTERVAL '130' MONTH (3)	130 months
INTERVAL '27' YEAR	27 years
INTERVAL '2-1' YEAR TO MONTH	Two years and one month
INTERVAL '100-1' YEAR(3) TO MONTH	100 years and one month

interval_day_to_second_literal::=

**Notes:**

- int defines the number of days (see precision for the maximal number of digits).
- time_expr specifies a time value in format HH[:MI[:SS[.n]]] or MI[:SS[.n]] or SS[.n]. Valid values are 0-23 for hours (HH), 0-59 for minutes (MI) and 0-59.999 for seconds (SS). The parameter precision defines the maximal number of digits for the leading field (see below) which allows you also to use larger numbers for hours, minutes and seconds (see examples). The parameter fractional_precision defines at which position the fractional part of the seconds are rounded (0-9, default is 3). In 'time_expr' you have to specify the whole used range, e.g. 'HH:MI' in case of HOUR TO MINUTE .
- Please mention that the precision of seconds is limited to 3 decimal places (like for timestamp values), although you can specify more digits in the literal.
- The optional parameter precision (1-9) specifies the maximal number of digits for the leading field. In default case, two digits are allowed.

- The interval borders must be descendant. That is why e.g. SECOND TO MINUTE is not valid.
- Examples:

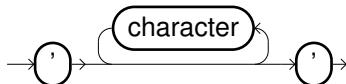
INTERVAL '5' DAY	Five days
INTERVAL '100' HOUR(3)	100 hours
INTERVAL '6' MINUTE	Six minutes
INTERVAL '1.99999' SECOND(2,2)	2.00 seconds (rounded after the second place)
INTERVAL '10:20' HOUR TO MINUTE	Ten hours and 20 minutes
INTERVAL '2 23:10:59' DAY TO SECOND	Two days, 23 hours, 10 minutes and 59 seconds
INTERVAL '23:10:59.123' HOUR(2) TO SECOND(3)	23 hours, 10 minutes and 59.123 seconds

2.5.5. String literals

String literals are used to represent text. The maximum length of a string is limited by the CHAR data type (see [String data types](#)).

The smallest character set is automatically used for a string literal. If only ASCII characters are included, then the ASCII character set will be used, otherwise the UTF8 character set (Unicode).

string_literal ::=



An empty string literal (' ') is evaluated as NULL value.



In order to use single quotes within a string, two single quotes are written next to one another. For example, the literal 'AB''C' represents the value AB'C.

2.5.6. NULL literal

The NULL literal is used in SQL to indicate a value as being "not known".

null_literal ::=



2.6. Format models

Format models are string literals, which represent a kind of conversion rule between strings and numeric or date/time values. Thus it is possible, for example, to assign a string to a numeric column or output a date as a string in a desired format.

2.6.1. Date/Time format models

Date/Time format models can be used in the functions [TO_CHAR \(datetime\)](#)/ [TO_DATE](#)/ [TO_TIMESTAMP](#) and in ETL commands [IMPORT](#)/ [EXPORT](#).

If no format is specified, the current default format is used. This is defined in the [NLS_DATE_FORMAT](#) and [NLS_TIMESTAMP_FORMAT](#) session parameters and can be changed for the current session by means of [ALTER SESSION](#) or for the entire database by means of [ALTER SYSTEM](#).

The default format is also important to implicit conversions (see also [Section 2.3.4, “Type conversion rules”](#)). If, for example, a string value is inserted into a date column, a [TO_DATE](#) is executed implicitly.

If no language is specified, the current session language is used for representing the date. This is defined in the [NLS_DATE_LANGUAGE](#) session parameter and can be changed for the current session by means of [ALTER SESSION](#) or for the entire database by means of [ALTER SYSTEM](#).

The current values for the NLS parameters can be found in the [EXA_PARAMETERS](#) system table.

For abbreviated month and day formats and those written in full, representation of the first two letters can be varied via use of upper case and lower case letters (see the examples on the next page).

The following table lists all the elements that can occur in date/time format models. It is important to note that each of the elements may only appear once in the format model. Exceptions to this are separators, padding characters and the output format for [TO_CHAR \(datetime\)](#). It should also be noted that in the English language setting abbreviated weekdays are displayed with three letters, but only two with the German.

Table 2.5. Elements of Date/Time format models

Element	Meaning
- ; . / \ _	Serve as separators or padding characters
CC	Century (01-99)
I Y Y Y, v Y Y Y and IYY, vYY, IY, vY, I, v	Year (0001-9999 in accordance with international standard, ISO 8601) and the last 3,2 or 1 digits
YYYY and YYYY, YY, Y	Year (0001-9999) and the last 3,2 or 1 digits
VYYY and VYY, VY, V	Year corresponding to the element VW, and the last 3,2 or 1 digits
Q	Quarter of the year (1-4)
MM	Month (01-12)
MON	Month abbreviation (JAN-DEC (ENG) or JAN-DEZ (DEU))
MONTH	Month written in full (JANUARY-DECEMBER (ENG) or JANUAR-DEZEMBER (DEU))
IW, vW	Week in the year (01-53, Mo-Su, week 01 of the year contains the 4th of January, previous days count to previous year, in accordance with international standard, ISO 8601)
uW	Week in the year (00-53, Mo-Su, week 01 of the year contains the 4th of January, previous days count to week 00)
VW	Week in the year (01-53, Su-Sa, week 01 of the year starts with the first sunday, previous days count to previous year)
UW	Week in the year (00-53, Su-Sa, week 01 of the year starts with the first sunday, previous days count to week 00)
WW	Week in the year (01-53, the first day of the year is the beginning of the week)
J	Julian Date (number of days since January 1, 4713 BC)
D	Day of week (1-7, starting from the day specified by the parameter NLS_FIRST_DAY_OF_WEEK - see also ALTER SYSTEM)
ID	ISO day of week (1-7, starting from Monday)
DD	Day (01-31)
DDD	Day in the year (001-365 or 001-366)
DAY	Weekday written in full (MONDAY-SUNDAY (ENG) or MONTAG-SONNTAG (DEU))
DY	Weekday abbreviation (MON-SUN (ENG) or MO-SO (DEU))
HH12	Hour (01-12)
HH24 (or HH)	Hour (00-23)
AM, A.M., am, a.m.	Meridian indicator
PM, P.M., pm, p.m.	Similar to AM
MI	Minute (00-59)
SS	Second (00-59)
FF[1-9]	Fraction of a second (e.g. milliseconds at FF3)

Examples

```
-- Is interpreted as 1 February 2003
SELECT TO_DATE('01-02-2003', 'DD-MM-YYYY');

-- Is interpreted as 10 February 2003
```

```

SELECT TO_DATE('06-2003-MON', 'WW-YYYY-DY');

-- Is interpreted as 31 December 2003, 12:59:33
SELECT TO_TIMESTAMP('12:59:33 365-2003', 'HH24:MI:SS DDD-YYYY');

-- Is interpreted as 24 December 2009, 23:00:00
SELECT TO_TIMESTAMP('2009-12-24 11:00:00 PM', 'YYYY-MM-DD HH12:MI:SS AM');

-- Is interpreted as 12 May 2003, 00:00:10.123
SELECT TO_TIMESTAMP('2000_MAY_12 10.123', 'YYYY_MONTH_DD SS.FF3');

SELECT TO_CHAR(DATE '2007-12-31', 'DAY-Q-DDD; IYYY\IW') TO_CHAR1,
       TO_CHAR(DATE '2000-01-01', 'DAY-Day-day-')           TO_CHAR2;

TO_CHAR1          TO_CHAR2
-----
MONDAY -4-365; 2008\01 SATURDAY -Saturday -saturday -

```

2.6.2. Numeric format models

Numeric format models specify the interpretation/presentation of strings/numbers and can be used in functions [TO_CHAR \(number\)](#)/[TO_NUMBER](#) and in ETL commands [IMPORT](#)/[EXPORT](#).

The following table lists all the elements that can occur in numeric format models.

Table 2.6. Elements of numeric format models

Element	Examples	Description
Digit		
9	9999,999	Each 9 stands for a digit of the number. If the number has fewer digits, it is padded with leading spacing characters.
0	0000,000	Each 0 stands for a digit of the number. If the number has fewer digits, it is padded with 0(s).
Sign		
S	S9999,999	Writes + for positive numbers and - for negative numbers. S may only be used at the beginning or end of the format string.
MI	9999,999MI	Writes a spacing character for positive numbers and - for negative numbers. MI may only be used at the end of the format string.
FM	FM9999,999	Removes all superfluous zeros. Writes - for negative numbers and removes all leading and trailing spaces for positive numbers. FM may only be used at the beginning of the format string.
Decimal separators		
.	999.999	Writes a decimal point before the fractional part.
D	999D999	Writes the decimal separator from NLS_NUMERIC_CHARACTERS before the fractional part.
Group separator		
,	999,999.999	Writes a comma as a group separator at the specified position.
G	999G999D999	Writes the group separator from NLS_NUMERIC_CHARACTERS at the specified position.
Others		
eeee	9.99eeee	Scientific notation.
EEEE	9.99EEEE	Scientific notation.

Comments

- If the format string is too short for the number, a string is returned, which is filled out with the character, #.
- The group and decimal separators are defined in the [NLS_NUMERIC_CHARACTERS](#) parameter and can be changed for the current session by means of [ALTER SESSION](#) or for the entire database by means of [ALTER SYSTEM](#).
- A point is used for the decimal separator and a comma for the group separator by default. The current values for the NLS parameters can be found in the [EXA_PARAMETERS](#) system table.

The following examples illustrate the use of numeric format models in function [TO_CHAR \(number\)](#).

Number	Format	Result	Description
123	99	'###'	The number is too long for the format
123	999	' 123'	Standard format for positive integers
-123	999	'-123'	Standard format for negative integers
123.456	999.999	' 123.456'	Standard format for positive decimals
-123.456	999.999	'-123.456'	Standard format for negative decimals
123.456	0000.0000	' 0123.4560'	Format filled with zeros
123.456	S999.999	'+123.456'	Explicit sign
123.456	999.999MI	'123.456 '	Sign at the end

Number	Format	Result	Description
-123.456	999.999MI	'123.456-'	Sign at the end
123.456	FM999.999	'123.456'	Removes trailing spaces and zeros
123456.789	999,999.999	' 123,456.789'	Group separator
123.4567	9.99eeee	'1.23e2'	Scientific notation
123.4567	9.99EEEE	'1.23E2'	Scientific notation

2.7. Operators

Operators combine one or two values (operands), e.g. in case of an addition of two numbers.

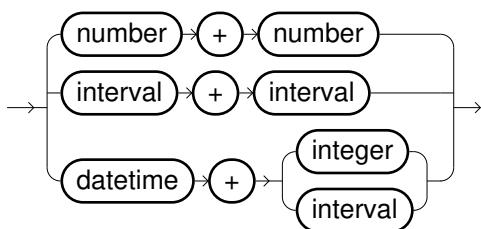
The precedence, that is the order in which the operators are evaluated, is the following:

+,-, PRIOR, CONNECT_BY_ROOT	As unary operators (e.g. in case of a negative number)
	Concatenation operator
*,/	Multiplication and division
+, -	Binary operators addition and subtraction

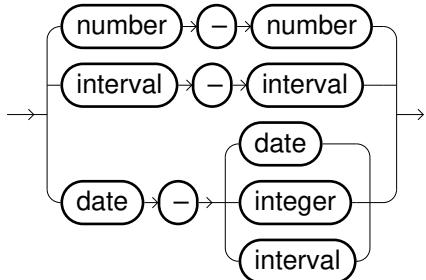
2.7.1. Arithmetic Operators

Syntax

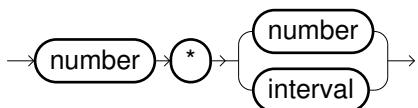
+ operator ::=



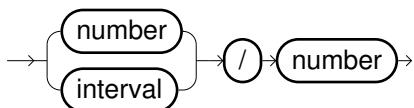
- operator ::=



* operator ::=



/ operator ::=



Note(s)

- + operator
 - If you add a decimal value to a timestamp, the result depends on the session/system parameter `TIMESTAMP_ARITHMETIC_BEHAVIOR`:

- `TIMESTAMP_ARITHMETIC_BEHAVIOR = 'INTERVAL'` - the decimal is rounded to an integer, so actually a certain number of full days is added (similar to function [ADD_DAYS](#))
- `TIMESTAMP_ARITHMETIC_BEHAVIOR = 'DOUBLE'` - the fraction of days is added (hours, minutes, ...)
- When adding a number of years or months (e.g. `INTERVAL '3' MONTH`) on a date whose day is the last day of the month, then the last day of the resulting month is returned (similar to [ADD_MONTHS](#))
- - operator
 - The difference of two datetime values depends on the session/system parameter `TIMESTAMP_ARITHMETIC_BEHAVIOR`:
 - `TIMESTAMP_ARITHMETIC_BEHAVIOR = 'INTERVAL'` - the result is an interval (similar to function [DAYS_BETWEEN](#))
 - `TIMESTAMP_ARITHMETIC_BEHAVIOR = 'DOUBLE'` - the result is a double
 - The difference of a date and a decimal is similar to the + operator with negative value, thus subtracting days instead of adding them.
 - The difference of two interval values return an interval
 - When subtracting a number of years or months (e.g. `INTERVAL '3' MONTH`) on a date whose day is the last day of the month, then the last day of the resulting month is returned (similar to [ADD_YEARS](#) and [ADD_MONTHS](#))

Example(s)

```

SELECT 1000 + 234                                ADD1,
        DATE '2000-10-05' + 1                      ADD2,
        DATE '2009-01-05' + INTERVAL '1-2' YEAR TO MONTH ADD3,
        100000 - 1                                 SUB1,
        DATE '2000-10-05' - 1                      SUB2,
        DATE '2009-01-05' - INTERVAL '2' MONTH       SUB3,
        100 * 123                                  MUL,
        100000 / 10                                DIV;

ADD1    ADD2        ADD3        SUB1      SUB2        SUB3        MUL      DIV
-----  -----  -----
1234   2000-10-06 2010-03-05  99999  2000-10-04  2008-11-05  12300  10000.0

```

2.7.2. Concatenation operator ||

Purpose

Returns the concatenation from `string1` and `string2`. This concatenation operator is equivalent to the [CONCAT](#) function.

Syntax

`||` operator ::=



Example(s)

```

SELECT 'abc' || 'DEF';

'abc' || 'DEF'

```

```
-----  
abcDEF
```

2.7.3. CONNECT BY Operators

Purpose

Details about the usage of operators can be found in the description of the [SELECT](#) statement in [Section 2.2.4, “Query language \(DQL\)”](#).

Syntax

PRIOR operator::=

→ **PRIOR** → **expr** →

CONNECT_BY_ROOT operator::=

→ **CONNECT_BY_ROOT** → **expr** →

Example(s)

```
SELECT last_name,  
       PRIOR last_name PARENT_NAME,  
       CONNECT_BY_ROOT last_name ROOT_NAME,  
       SYS_CONNECT_BY_PATH(last_name, '/') "PATH"  
  FROM employees  
 CONNECT BY PRIOR employee_id = manager_id  
 START WITH last_name = 'Clark';  
  
LAST_NAME PARENT_NAME ROOT_NAME PATH  
-----  
Clark          Clark      /Clark  
Smith         Clark      /Clark/Smith  
Brown         Smith     /Clark/Smith/Brown
```

2.8. Predicates

2.8.1. Introduction

Predicates are expressions which return a boolean value as the result, i.e. FALSE, TRUE, or NULL (or its alias, UNKNOWN).

Predicates can be used as follows:

- in the SELECT list as well as in the WHERE and HAVING clauses of a SELECT query
- in the WHERE clause of the [UPDATE](#) and [DELETE](#) statements
- in the ON clause and the WHERE clauses of the [MERGE](#) statement

The following predicates are currently supported:

[Comparison predicates](#)

[Logical join predicates](#)

[NOT] BETWEEN

EXISTS

[NOT] IN

IS [NOT] NULL

[NOT] REGEXP_LIKE

[NOT] LIKE

The order in which predicates are evaluated in complex expressions is determined by the precedence of the respective predicate. The following table defines this in descending order, i.e. the predicates of the first row will be evaluated first. However, the desired evaluation sequence can be specified by enclosing the expressions in parentheses.

Table 2.7. Precedence of predicates

Predicates	Designation
=, !=, <, <=, >, >=	Comparison predicates
[NOT] BETWEEN, EXISTS, [NOT] IN, IS [NOT] NULL, [NOT] REGEXP_LIKE, [NOT] LIKE	Special predicates
NOT	Logical negation
AND	Conjunction
OR	Disjunction

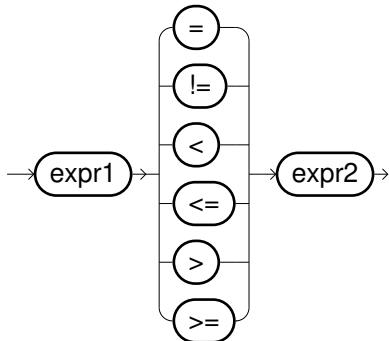
2.8.2. List of predicates

Comparison predicates

Purpose

Comparison predicates compare two expressions and return whether the comparison is true.

Syntax



Note(s)

=	Parity check
!=	Disparity check (the aliases \neq and $\wedge\neq$ also exist for this)
< or <=	Check for "less than" or "less than or equal to"
> or >=	Check for "more than" or "more than or equal to"
NULL values	If one of the two expressions is the NULL value, the result is also the NULL value.

Example(s)

```

SELECT 1=1, 1<1, 1!=null;

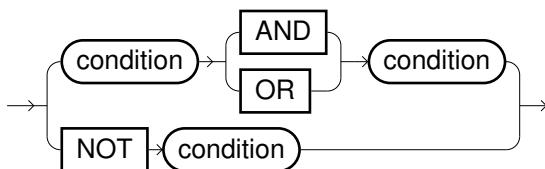
1=1      1<1      1<>NULL
----- -----
TRUE     FALSE
  
```

Logical join predicates

Purpose

Join predicates (also known as Boolean operators) join Boolean values.

Syntax



Note(s)

The following tables define the behavior of join operators:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE

NULL	NULL	FALSE	NULL
OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL
	TRUE	FALSE	NULL
NOT	FALSE	TRUE	NULL

Example(s)

```
SELECT true AND false AS res1,
      NOT false      AS res2,
      true OR null   AS res3;
```

RES1	RES2	RES3

FALSE	TRUE	TRUE

[NOT] BETWEEN**Purpose**

Tests whether an expression is situated between two values.

Syntax**Note(s)**

- A BETWEEN B AND C is equivalent to B <= A AND A <= C.

Example(s)

```
SELECT 2 BETWEEN 1 AND 3 AS res;

RES
-----
TRUE
```

EXISTS**Purpose**

Tests whether the specified subquery contains result rows.

Syntax



Example(s)

```

CREATE TABLE t (i DECIMAL);
INSERT INTO t VALUES 1,2;
CREATE TABLE t2 (j DECIMAL);
INSERT INTO t2 VALUES 2;

SELECT i FROM t WHERE EXISTS (SELECT * FROM t2);

i
-----
1
2

SELECT i FROM t WHERE EXISTS (SELECT * FROM t2 WHERE t.i=t2.j);

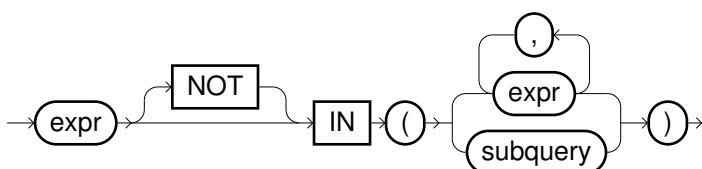
i
-----
2
  
```

[NOT] IN

Purpose

Tests whether an expression is contained in a result set.

Syntax



Example(s)

```

CREATE TABLE t (x DECIMAL);
INSERT INTO t VALUES 1,2,3,4;

SELECT x FROM t WHERE x IN (2,4);

x
-----
2
4

CREATE TABLE t2 (y DECIMAL);
  
```

```
INSERT INTO t2 VALUES 2,4;

SELECT x FROM t WHERE x IN (SELECT y FROM t2);

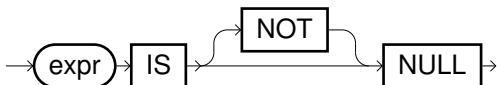
x
-----
2
4
```

IS [NOT] NULL

Purpose

Tests whether an expression is the NULL value.

Syntax



Example(s)

```
SELECT null=null, null IS NULL, '' IS NOT NULL;
NULL=NULL NULL IS NULL '' IS NOT NULL
-----
TRUE          FALSE
```

[NOT] REGEXP_LIKE

Purpose

This predicate states whether a string matches a regular expression.

Syntax



Note(s)

- Details and examples for regular expressions can be found in [Section 2.1.3, “Regular expressions”](#).
- See also functions [SUBSTR\[ING\]](#), [REGEXP_INSTR](#), [REGEXP_SUBSTR](#) and [REGEXP_REPLACE](#).

Example(s)

```
SELECT 'My mail address is my_mail@exasol.com'  
      REGEXP_LIKE '(?i).*[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}.*'  
      AS contains_email;  
  
CONTAINS_EMAIL  
-----  
TRUE
```

[NOT] LIKE

Purpose

Similar to the predicate [\[NOT\] REGEXP_LIKE](#), but only with simple pattern matching.

Syntax



Note(s)

- Special characters:
 - _ Wildcard for exactly one character
 - % Wildcard for any number of characters (including no characters at all)
 - esc_chr Character with which the characters _ and % can be used literally in a pattern. By default this is the session variable DEFAULT_LIKE_ESCAPE_CHARACTER (see also [ALTER SESSION](#)).

Example(s)

```
SELECT 'abcd' LIKE 'a_d' AS res1, '%bcd' like '\%%d' AS res2;  
  
RES1  RES2  
-----  
FALSE TRUE
```

2.9. Built-in functions

The chapter documents the functions supported by Exasol. They should not be confused with user-defined functions, which can be created by means of [CREATE FUNCTION](#).

The individual sections summarize the functions according to their use. The actual description of the functions can be found in [Section 2.9.4, “Alphabetical list of all functions”](#).

2.9.1. Scalar functions

Scalar functions receive an input value and from this deliver a result value. They can be used with constant values, the column elements of a table (e.g. view) as well as with compound value expressions.

Examples:

```
SELECT SIN(1);
SELECT LENGTH(s) FROM t;
SELECT EXP(1+ABS(n)) FROM t;
```

Scalar functions normally expect a special data type for their arguments. If this is not specified, an implicit conversion to this data type is attempted or an error message is issued.

Numeric functions

Numeric functions are given a numeric value as input and normally deliver a numeric value as output.

[ABS](#)
[ACOS](#)
[ASIN](#)
[ATAN](#)
[ATAN2](#)
[CEIL\[ING\]](#)
[COS](#)
[COSH](#)
[COT](#)
[DEGREES](#)
[DIV](#)
[EXP](#)
[FLOOR](#)
[LN](#)
[LOG](#)
[LOG10](#)
[LOG2](#)
[MOD](#)
[PI](#)
[POWER](#)
[RADIAN](#)
[RAND\[OM\]](#)
[ROUND \(number\)](#)
[SIGN](#)
[SIN](#)
[SINH](#)
[SQRT](#)
[TAN](#)
[TANH](#)

[TO_CHAR \(number\)](#)
[TO_NUMBER](#)
[TRUNC\[ATE\] \(number\)](#)

String functions

String functions can either return a string (e.g. LPAD) or a numeric value (e.g. LENGTH).

[ASCII](#)
[BIT_LENGTH](#)
[CHARACTER_LENGTH](#)
[CH\[A\]R](#)
[COLOGNE_PHONETIC](#)
[CONCAT](#)
[DUMP](#)
[EDIT_DISTANCE](#)
[INSERT](#)
[INSTR](#)
[LCASE](#)
[LEFT](#)
[LENGTH](#)
[LOCATE](#)
[LOWER](#)
[LPAD](#)
[LTRIM](#)
[MID](#)
[OCTET_LENGTH](#)
[POSITION](#)
[REGEXP_INSTR](#)
[REGEXP_REPLACE](#)
[REGEXP_SUBSTR](#)
[REPEAT](#)
[REPLACE](#)
[REVERSE](#)
[RIGHT](#)
[RPAD](#)
[RTRIM](#)
[SOUNDEX](#)
[SPACE](#)
[SUBSTR\[ING\]](#)
[TO_CHAR \(datetime\)](#)
[TO_CHAR \(number\)](#)
[TO_NUMBER](#)
[TRANSLATE](#)
[TRIM](#)
[UCASE](#)
[UNICODE](#)
[UNICODECHR](#)
[UPPER](#)

Date/Time functions

Date/Time functions manipulate the DATE, TIMESTAMP, TIMESTAMP WITH LOCAL TIME ZONE and INTERVAL data types.

[ADD_DAYS](#)
[ADD_HOURS](#)

ADD_MINUTES
ADD_MONTHS
ADD_SECONDS
ADD_WEEKS
ADD_YEARS
CONVERT_TZ
CURDATE
CURRENT_DATE
CURRENT_TIMESTAMP
DATE_TRUNC
DAY
DAYS_BETWEEN
DBTIMEZONE
EXTRACT
FROM_POSIX_TIME
HOUR
HOURS_BETWEEN
LOCALTIMESTAMP
MINUTE
MINUTES_BETWEEN
MONTH
MONTHS_BETWEEN
NOW
NUMTODSINTERVAL
NUMTOYMINTERVAL
POSIX_TIME
ROUND (datetime)
SECOND
SECONDS_BETWEEN
SESSIONTIMEZONE
SYSDATE
SYSTIMESTAMP
TO_CHAR (datetime)
TO_DATE
TO_DSINTERVAL
TO_TIMESTAMP
TO_YMINTERVAL
TRUNC[ATE] (datetime)
WEEK
YEAR
YEARS_BETWEEN

Geospatial functions

There exist a lot of functions to analyze geospatial data (see [ST_*](#) and [Section 2.4, “Geospatial data”](#)).

Bitwise functions

Bitwise functions can compute bit operations on numerical values.

BIT_AND
BIT_CHECK
BIT_LROTATE
BIT_LSHIFT
BIT_NOT
BIT_OR

BIT_RROTATE
BIT_RSHIFT
BIT_SET
BIT_TO_NUM
BIT_XOR

Conversion functions

Conversion functions can be used to convert values to other data types.

CAST
CONVERT
IS_*
NUMTODSINTERVAL
NUMTOYMINTERVAL
TO_CHAR (datetime)
TO_CHAR (number)
TO_DATE
TO_DSINTERVAL
TO_NUMBER
TO_TIMESTAMP
TO_YMINTERVAL

Functions for hierarchical queries

The following functions can be used in combination with CONNECT BY queries.

CONNECT_BY_ISCYCLE
CONNECT_BY_ISLEAF
LEVEL
SYS_CONNECT_BY_PATH

Other scalar functions

Those functions that cannot be allocated to one of the above categories are listed here.

CASE
COALESCE
CURRENT_SCHEMA
CURRENT_SESSION
CURRENT_STATEMENT
CURRENT_USER
DECODE
GREATEST
HASH_MD5
HASH_SHA[1]
HASH_TIGER
IPROC
LEAST
NULLIF
NULLIFZERO
NPROC
NVL
NVL2
ROWID
SYS_GUID

[USER](#)[VALUE2PROC](#)[ZEROIFNULL](#)

2.9.2. Aggregate functions

An aggregate function refers to a set of input values and returns one single result value. If the table is subdivided into several groups with the GROUP BY clause, an aggregate function calculates a value for each of these groups.

If a GROUP BY clause is not stated, an aggregate function always refers to the entire table. This type of query then returns exactly one result row.

Aggregate functions are sometimes referred to as *set functions*.

[APPROXIMATE_COUNT_DISTINCT](#)[AVG](#)[CORR](#)[COUNT](#)[COVAR_POP](#)[COVAR_SAMP](#)[FIRST_VALUE](#)[GROUP_CONCAT](#)[GROUPING\[_ID\]](#)[LAST_VALUE](#)[MAX](#)[MEDIAN](#)[MIN](#)[PERCENTILE_CONT](#)[PERCENTILE_DISC](#)[REGR_*](#)[ST_INTERSECTION \(see \[ST_*\]\(#\) and Section 2.4, “Geospatial data”\)](#)[ST_UNION \(see \[ST_*\]\(#\) and Section 2.4, “Geospatial data”\)](#)[STDDEV](#)[STDDEV_POP](#)[STDDEV_SAMP](#)[SUM](#)[VAR_POP](#)[VAR_SAMP](#)[VARIANCE](#)

2.9.3. Analytical functions

Analytical functions always refer to subsets of the data. These so-called partitions are defined by the PARTITION BY clause. If this is not stated, the analytical function refers to the whole table. The ORDER BY clause can be used to specify the way in which the data within a partition should be sorted. Specification of the order is compulsory for many analytical functions because it is essential to the computation.

If an ORDER BY is specified for an analytical function, the number of data records relevant to the computation can be further restricted by defining a window (WINDOW). Normally the window encompasses the data records from the beginning of the partition to the current data record, however, it can also be explicitly restricted with ROWS (physical border). Restriction using RANGE (logical border) is not yet supported by Exasol.

With exception to ORDER BY, analytical functions are executed to the end, i.e. after evaluating the WHERE, GROUP BY, and HAVING clauses. Therefore, they may only be used in the SELECT list or the ORDER BY clause.

Analytical functions enable complex evaluations and analyzes through a variety of statistical functions and are a valuable addition to aggregate functions (see [Section 2.9.2, “Aggregate functions”](#)). The following analytical functions are supported:

```
AVG
CORR
COUNT
COVAR_POP
COVAR_SAMP
DENSE_RANK
FIRST_VALUE
LAG
LAST_VALUE
LEAD
MAX
MEDIAN
MIN
PERCENTILE_CONT
PERCENTILE_DISC
RANK
RATIO_TO_REPORT
REGR_*
ROW_NUMBER
STDDEV
STDDEV_POP
STDDEV_SAMP
SUM
VAR_POP
VAR_SAMP
VARIANCE
```

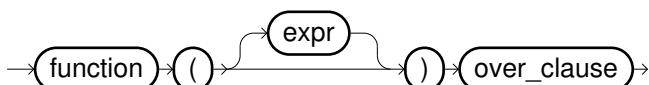
Analytical query

Purpose

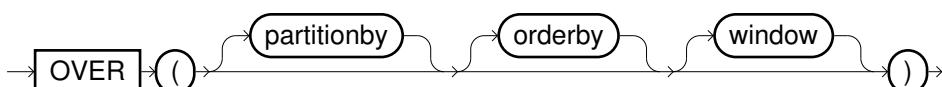
Analytical functions evaluate a set of input values. However, unlike aggregate functions they return a result value for each database row and not for each group of rows.

Syntax

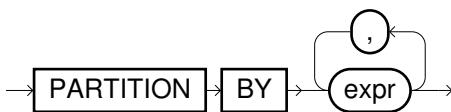
analytical_query :=



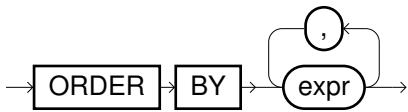
over_clause :=



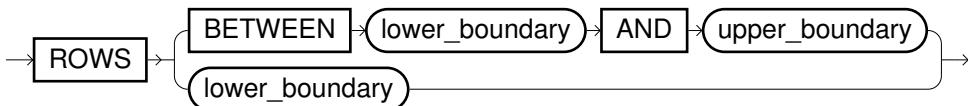
partitionby :=



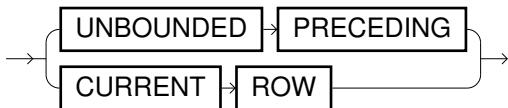
orderby :=



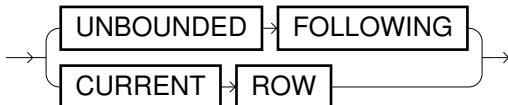
window :=



lower_boundary :=



upper_boundary :=



Note(s)

- Analytical functions are always evaluated after WHERE, GROUP BY and HAVING but before ORDER BY (global).
- If the table is divided into multiple partitions via the PARTITION BY clause, the results within each partition are calculated independently from the rest of the table. If no PARTITION BY clause is stated, an analytical function always refers to the entire table.
- A reliable collation of rows for an analytical function can only be achieved using ORDER BY *within* the OVER - clause.
- If an ORDER BY clause is used, you can additionally limit the set of relevant rows for the computation by specifying a window. The default window is ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

Example(s)

```

SELECT age,
       FLOOR(age/10) || '0ies' AS agegroup,
       COUNT(*) OVER (
           PARTITION BY FLOOR(age/10)
           ORDER BY age
       ) AS COUNT
FROM staff;

AGE   AGEGROUP  COUNT
-----
```

25	20ies	1
26	20ies	2
27	20ies	3
28	20ies	4
31	30ies	1
39	30ies	2

2.9.4. Alphabetical list of all functions

ABS

Purpose

Returns the absolute sum of number n.

Syntax

abs::=

→ **ABS** → (→ n →) →

Example(s)

```
SELECT ABS(-123) ABS;  
  
ABS  
----  
123
```

ACOS

Purpose

Returns the arccosine of number n. The result is between 0 and π .

Syntax

acos::=

→ **ACOS** → (→ n →) →

Note(s)

- Number n must be in the interval [-1;1].

Example(s)

```
SELECT ACOS(0.5) ACOS;
```

```
ACOS  
-----  
1.047197551196598
```

ADD_DAYS

Purpose

Adds a specified number of days to a date or timestamp.

Syntax

add_days::=

→ **ADD_DAYS** → (→ **datetime** → , → **integer** →) →

Note(s)

- Decimals are rounded before adding that number.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT ADD_DAYS(DATE '2000-02-28', 1) AD1,  
       ADD_DAYS(TIMESTAMP '2001-02-28 12:00:00', 1) AD2;  
  
AD1          AD2  
-----  -----  
2000-02-29  2001-03-01 12:00:00.000000
```

ADD_HOURS

Purpose

Adds a specified number of hours to a timestamp.

Syntax

add_hours::=

→ **ADD_HOURS** → (→ **datetime** → , → **integer** →) →

Note(s)

- Decimals are rounded before adding that number.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated internally within UTC before the result is transformed to the session time zone.

Example(s)

```
SELECT ADD_HOURS(TIMESTAMP '2000-01-01 00:00:00', 1) AH1,
       ADD_HOURS(TIMESTAMP '2000-01-01 12:23:45', -1) AH2;
```

AH1	AH2
-----	-----
2000-01-01 01:00:00.000000	2000-01-01 11:23:45.000000

ADD_MINUTES

Purpose

Adds a specified number of minutes to a timestamp.

Syntax

add_minutes ::=

→ **ADD_MINUTES** → (→ **datetime** → , → **integer** →) →

Note(s)

- Decimals are rounded before adding that number.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated internally within UTC before the result is transformed to the session time zone.

Example(s)

```
SELECT ADD_MINUTES(TIMESTAMP '2000-01-01 00:00:00', -1) AM1,
       ADD_MINUTES(TIMESTAMP '2000-01-01 00:00:00', +2) AM2;
```

AM1	AM2
-----	-----
1999-12-31 23:59:00.000000	2000-01-01 00:02:00.000000

ADD_MONTHS

Purpose

Adds a specified number of months to a date or timestamp.

Syntax

add_months ::=

→ **ADD_MONTHS** → (→ **datetime** → , → **integer** →) →

Note(s)

- Decimals are rounded before adding that number.
- If the resulting month has fewer days than the day of the date of entry, the last day of this month is returned.
- If the input date is the last day of a month, then the last day of the resulting month is returned.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT ADD_MONTHS(DATE '2006-01-31', 1) AM1,
       ADD_MONTHS(TIMESTAMP '2006-01-31 12:00:00', 2) AM2;

AM1          AM2
-----
2006-02-28  2006-03-31 12:00:00.000000
```

ADD_SECONDS

Purpose

Adds a specified number of seconds to a timestamp.

Syntax

add_seconds::=

→ **ADD_SECONDS** → (→ **datetime** → , → **decimal** →) →

Note(s)

- Up to three digits after the decimal point can be processed.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated internally within UTC before the result is transformed to the session time zone.

Example(s)

```
SELECT ADD_SECONDS(TIMESTAMP '2000-01-01 00:00:00', -1) AS1,
       ADD_SECONDS(TIMESTAMP '2000-01-01 00:00:00', +1.234) AS2;

AS1          AS2
-----
1999-12-31 23:59:59.000000  2000-01-01 00:00:01.234000
```

ADD_WEEKS

Purpose

Adds a specified number of weeks to a date or timestamp.

Syntax

add_weeks::=



Note(s)

- ADD_WEEKS(x,n) is similar to ADD_DAYS(x,n*7).
- Decimals are rounded before adding that number.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT ADD_WEEKS(DATE '2000-02-29', 1) AW1,
       ADD_WEEKS(TIMESTAMP '2005-01-31 12:00:00', -1) AW2;

AW1          AW2
-----
2000-03-07  2005-01-24 12:00:00.000000
```

ADD_YEARS

Purpose

Adds a specified number of years to a date or timestamp.

Syntax

add_years::=



Note(s)

- Decimals are rounded before adding that number.
- If the resulting month has fewer days than the day of the date of entry, the last day of this month is returned.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT ADD_YEARS(DATE '2000-02-29', 1) AY1,
       ADD_YEARS(TIMESTAMP '2005-01-31 12:00:00', -1) AY2;

AY1          AY2
-----
2001-02-28  2004-01-31 12:00:00.000000
```

APPROXIMATE_COUNT_DISTINCT

Purpose

Returns the approximate number of distinct elements (without NULL).

Syntax

approximate_count_distinct::=

→ **APPROXIMATE_COUNT_DISTINCT** → (→ **expr** →) →

Note(s)

- The result isn't exact as it is with function **COUNT**, but it can be computed a lot faster.
- For the calculation, the algorithm HyperLogLog is used internally.

Example(s)

```
SELECT COUNT(DISTINCT customer_id) COUNT_EXACT,  
APPROXIMATE_COUNT_DISTINCT (customer_id) COUNT_APPR  
FROM orders WHERE price > 1000;  
  
COUNT_EXACT COUNT_APPR  
----- -----  
10000000 10143194
```

ASCII

Purpose

Returns the numeric value of a character in the ASCII character set.

Syntax

ascii::=

→ **ASCII** → (→ **char** →) →

Note(s)

- If the character is no ASCII character, then an exception is thrown.

Example(s)

```
SELECT ASCII( 'X' );  
  
ASCII( 'X' )  
-----  
88
```

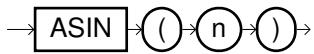
ASIN

Purpose

Returns the arcsine of number n. The result is between $-\pi/2$ and $\pi/2$.

Syntax

asin ::=



Note(s)

- Number n must be in the interval [-1;1].

Example(s)

```
SELECT ASIN(1);  
ASIN(1)  
-----  
1.570796326794897
```

ATAN

Purpose

Returns the arctangent of number n. The result is between $-\pi/2$ and $\pi/2$.

Syntax

atan ::=



Example(s)

```
SELECT ATAN(1);  
ATAN(1)  
-----  
0.785398163397448
```

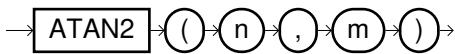
ATAN2

Purpose

Returns the arctangent of two numbers n and m. The expression is equivalent to $\text{ATAN}(n/m)$.

Syntax

atan2::=



Example(s)

```
SELECT ATAN2(1,1) ATAN2;  
  
ATAN2(1,1)  
-----  
0.785398163397448
```

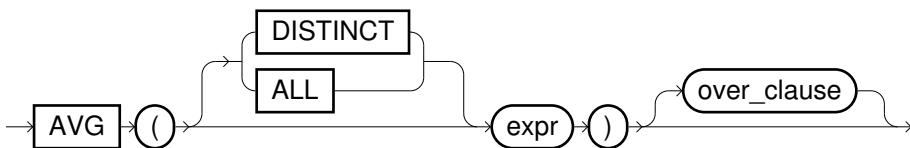
AVG

Purpose

Returns the mean value.

Syntax

avg::=



Note(s)

- If ALL or nothing is specified, then all of the entries are considered. If DISTINCT is specified, duplicate entries are only accounted for once.
- Only numeric operands are supported.

Example(s)

```
SELECT AVG(age) AVG FROM staff;  
  
AVG  
-----  
36.25
```

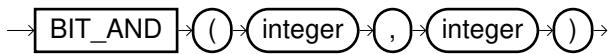
BIT_AND

Purpose

Computes the bitwise AND operation of two numerical values.

Syntax

bit_and ::=



Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The result data type is DECIMAL(20,0).

Example(s)

```
-- 1001 AND 0011 = 0001
SELECT BIT_AND(9,3);

BIT_AND(9,3)
-----
1
```

BIT_CHECK

Purpose

Checks whether a certain bit of a numerical value is set. The position parameter starts from 0 which means the lowest bit.

Syntax

bit_check ::=



Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The value pos may be between 0 and 63.

Example(s)

```
SELECT BIT_CHECK(3,0) B0,
       BIT_CHECK(3,1) B1,
       BIT_CHECK(3,2) B2,
       BIT_CHECK(3,3) B3;

B0      B1      B2      B3
----- -----
TRUE    TRUE    FALSE   FALSE
```

BIT_LENGTH

Purpose

Returns the bit length of a string. If only ASCII characters are used, then this function is equivalent to [CHARACTER_LENGTH](#) * 8.

Syntax

bit_length ::=

→ **BIT_LENGTH** → (→ **string** →) →

Example(s)

```
SELECT BIT_LENGTH( 'aou' )  BIT_LENGTH;
BIT_LENGTH
-----
24

SELECT BIT_LENGTH( 'äöü' )  BIT_LENGTH;
BIT_LENGTH
-----
48
```

BIT_LROTATE

Purpose

Rotates the bits of a number by the specified number to the left.

Syntax

bit_lrotate ::=

→ **BIT_LROTATE** → (→ **integer** → , → **integer** →) →

Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The second parameter can be between 0 and 63.
- The result data type is DECIMAL(20,0).

Example(s)

```
SELECT BIT_LROTATE(1024,63);
BIT_LROTATE(1024,63)
```

BIT_LSHIFT

Purpose

Shifts the bits of a number by the specified number to the left.

Syntax

bit_lshift::=

→ **BIT_LSHIFT** → (→ integer → , → integer →) →

Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The second parameter can be between 0 and 63.
- The result data type is DECIMAL(20,0).

Example(s)

```
SELECT BIT_LSHIFT(1,10);  
  
BIT_LSHIFT(1,10)  
-----  
1024
```

BIT_NOT

Purpose

Computes the bitwise negation of a single numerical value.

Syntax

bit_not::=

→ **BIT_NOT** → (→ integer →) →

Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The result data type is DECIMAL(20,0).

Example(s)

```
SELECT BIT_NOT(0), BIT_NOT(18446744073709551615);

BIT_NOT(0)          BIT_NOT(18446744073709551615)
-----
18446744073709551615          0

SELECT BIT_AND( BIT_NOT(1), 5 );

BIT_AND(BIT_NOT(1),5)
-----
4
```

BIT_OR

Purpose

Computes the bitwise OR operation of two numerical values.

Syntax

bit_or::=

→ **BIT_OR** → (→ integer → , → integer →) →

Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The result data type is DECIMAL(20,0).

Example(s)

```
-- 1001 OR 0011 = 1011
SELECT BIT_OR(9,3);

BIT_OR(9,3)
-----
11
```

BIT_RROTATE

Purpose

Rotates the bits of a number by the specified number to the right.

Syntax

bit_rrotate::=

→ **BIT_RROTATE** → (→ integer → , → integer →) →

Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The second parameter can be between 0 and 63.
- The result data type is DECIMAL(20,0).

Example(s)

```
SELECT BIT_RROTRATE(1024,63);  
  
BIT_RROTRATE(1024,63)  
-----  
2048
```

BIT_RSHIFT

Purpose

Shifts the bits of a number by the specified number to the right.

Syntax

bit_rshift::=

→ **BIT_RSHIFT** → (→ integer → , → integer →) →

Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The second parameter can be between 0 and 63.
- The result data type is DECIMAL(20,0).

Example(s)

```
SELECT BIT_RSHIFT(1024,10);  
  
BIT_RSHIFT(1024,10)  
-----  
1
```

BIT_SET

Purpose

Sets a certain bit of a numerical value. The position parameter starts from 0 which means the lowest bit.

Syntax

bit_set::=



Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The value pos may be between 0 and 63.
- The result data type is DECIMAL(20,0).

Example(s)

```
SELECT BIT_SET(8,0);
BIT_SET(8,0)
-----
9
```

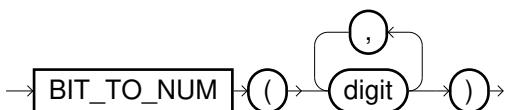
BIT_TO_NUM

Purpose

Creates a numerical value out of a list of single bits.

Syntax

bit_to_num ::=



Note(s)

- Each function argument must evaluate to 0 or 1.
- The first argument is interpreted as the biggest bit.
- The number of arguments is limited to 64, which means you can create positive numbers between 0 and 18446744073709551615.

Example(s)

```
SELECT BIT_TO_NUM(1,1,0,0);
BIT_TO_NUM(1,1,0,0)
-----
12
```

BIT_XOR

Purpose

Computes the bitwise exclusive OR operation of two numerical values. The result in each position is 1 if the two corresponding bits are different.

Syntax

bit_xor ::=



Note(s)

- Bit functions are limited to 64 bits, which means to positive numbers between 0 and 18446744073709551615.
- The result data type is DECIMAL(20,0).

Example(s)

```
-- 1001 XOR 0011 = 1010
SELECT BIT_XOR(9,3);

BIT_XOR(9,3)
-----
10
```

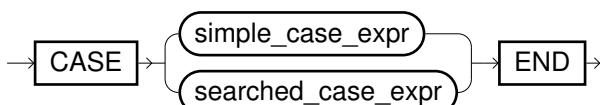
CASE

Purpose

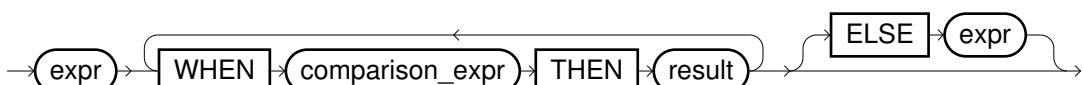
With the aid of the CASE function, an IF THEN ELSE logic can be expressed within the SQL language.

Syntax

case ::=



simple_case_expr ::=



searched_case_expr ::=



Note(s)

- With the `simple_case_expr` the `expr` is compared with the specified alternatives. The `THEN` part of the first match defines the result.
- With the `searched_case_expr` the row is evaluated using all of the conditions until one equates to the `TRUE` value. The `THEN` part of this condition is the result.
- If none of the options apply, the `ELSE` value is returned. If this was not specified, the `NULL` value is returned.

Example(s)

```

SELECT name, CASE grade WHEN 1 THEN 'VERY GOOD'
                      WHEN 2 THEN 'GOOD'
                      WHEN 3 THEN 'SATISFACTORY'
                      WHEN 4 THEN 'FAIR'
                      WHEN 5 THEN 'UNSATISFACTORY'
                      WHEN 6 THEN 'POOR'
                      ELSE 'INVALID'
END AS GRADE FROM student;

NAME      GRADE
-----
Fischer  VERY GOOD
Schmidt   FAIR

SELECT name, CASE WHEN turnover>1000 THEN 'PREMIUM'
                  ELSE 'STANDARD'
END AS CLASS FROM customer;

NAME      CLASS
-----
Meier    STANDARD
Huber    PREMIUM

```

CAST**Purpose**

Converts an expression into the specified data type. If this is not possible, then an exception is thrown.

Syntax

`cast::=`

→ **CAST** → (→ **expr** → **AS** → **data_type** →) →

Note(s)

- `CONVERT` is an alias for this function.

Example(s)

```
SELECT CAST( 'ABC' AS CHAR(15)) STRINGCAST;
STRINGCAST
-----
ABC

SELECT CAST( '2006-01-01' AS DATE) DATECAST;
DATECAST
-----
2006-01-01
```

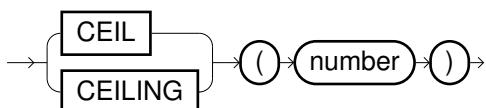
CEIL[ING]

Purpose

Returns the smallest whole number that is larger or equal to the given number.

Syntax

ceiling ::=



Example(s)

```
SELECT CEIL(0.234) CEIL;
CEIL
-----
1
```

CHARACTER_LENGTH

Purpose

Returns the length of a string in characters.

Syntax

character_length ::=



Example(s)

```
SELECT CHARACTER_LENGTH( 'aeiouäöü' ) C_LENGTH;
```

```
C_LENGTH
```

```
-----
```

```
8
```

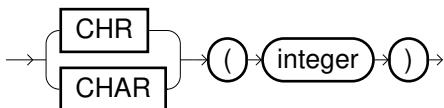
CH[A]R

Purpose

Returns the ASCII character whose ordinal number is the given integer.

Syntax

chr ::=



Note(s)

- The number n must be between 0 and 127.
- CHR(0) returns NULL.

Example(s)

```
SELECT CHR( 88 )  CHR;
```

```
CHR
```

```
---
```

```
X
```

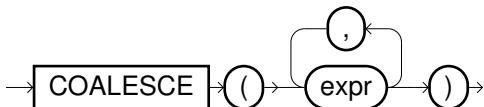
COALESCE

Purpose

Returns the first value from the argument list which is not NULL. If all of the values are NULL, the function returns NULL.

Syntax

coalesce ::=



Note(s)

- The minimum number of arguments is 2.

- The COALESCE(expr1,expr2) function is equivalent to the CASE expression CASE WHEN expr1 IS NOT NULL THEN expr1 ELSE expr2 END

Example(s)

```
SELECT COALESCE(NULL, 'abc',NULL, 'xyz') COALES;
COALES
-----
abc
```

COLOGNE_PHONETIC

Purpose

This function returns a phonetic representation of a string. You can use it to compare words which sounds similar, but are spelled different.

Syntax

cologne_phonetic::=

→ **COLONE_PHONETIC** → (→ **string** →) →

Note(s)

- A description for the used algorithm can be found here: http://de.wikipedia.org/wiki/Kölner_Phonetik
- The result is a string of digits, whose length has as maximum the double length of the input string.
- This function is similar to **SOUNDEX**, but is more appropriate for German words.
- To calculate the difference between two strings, you can use the function **EDIT_DISTANCE**.

Example(s)

```
SELECT COLOGNE_PHONETIC('schmitt'), COLOGNE_PHONETIC('Schmidt');
CLOGNE_PHONETIC('schmitt') COLOGNE_PHONETIC('Schmidt')
-----
862 862
```

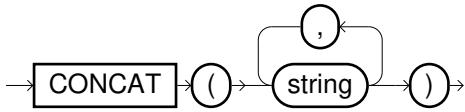
CONCAT

Purpose

Returns the concatenation of a number of strings.

Syntax

concat::=



Note(s)

- This function is equivalent to the [Section 2.7.2, “Concatenation operator ||”](#).

Example(s)

```
SELECT CONCAT( 'abc' , 'def' ) CONCAT;
CONCAT
-----
abcdef
```

CONNECT_BY_ISCYCLE

Purpose

Returns for a CONNECT BY query whether a row causes a cycle. Details can be found in the description of the [SELECT statement](#) in [Section 2.2.4, “Query language \(DQL\)”](#).

Syntax

connect_by_iscycle ::=



Example(s)

```
SELECT CONNECT_BY_ISCYCLE ,
       SYS_CONNECT_BY_PATH(last_name, '/') "PATH"
    FROM employees WHERE last_name = 'Clark'
CONNECT BY NOCYCLE PRIOR employee_id = manager_id
START WITH last_name = 'Clark';

CONNECT_BY_ISCYCLE PATH
-----
0 /Clark
1 /Clark/Jackson/Johnson/Clark
```

CONNECT_BY_ISLEAF

Purpose

Returns for a CONNECT BY query whether a row is a leaf within the tree. Details can be found in the description of the [SELECT statement](#) in [Section 2.2.4, “Query language \(DQL\)”](#).

Syntax

connect_by_isleaf ::=

→ **CONNECT_BY_ISLEAF** →

Example(s)

```
SELECT last_name, CONNECT_BY_ISLEAF,
       SYS_CONNECT_BY_PATH(last_name, '/') "PATH"
  FROM employees
 CONNECT BY PRIOR employee_id = manager_id
 START WITH last_name = 'Clark';

LAST_NAME CONNECT_BY_ISLEAF PATH
-----
Clark                  0 /Clark
Smith                 0 /Clark/Smith
Brown                 1 /Clark/Smith/Brown
Jones                 1 /Clark/Smith/Jones
```

CONVERT

Purpose

Converts an expression into the specified data type if this is possible.

Syntax

convert ::=

→ **CONVERT** → (→ **data_type** → , → **expr** →) →

Note(s)

- This function is an alias for [CAST](#).

Example(s)

```
SELECT CONVERT( CHAR(15), 'ABC' ) STRINGCAST;

STRINGCAST
-----
ABC

SELECT CONVERT( DATE, '2006-01-01' ) DATECAST;

DATECAST
-----
2006-01-01
```

CONVERT_TZ

Purpose

Converts a timestamp from one time zone into another one. Please note that timestamps don't contain any timezone information. This function therefore just adds the time shift between two specified timezones.

Syntax

convert_tz::=



Note(s)

- The list of supported timezones can be found in the system table [EXA_TIME_ZONES](#).
- If the input value has type TIMESTAMP WITH LOCAL TIME ZONE, then this function is only allowed if the session time zone ([SESSIONTIMEZONE](#)) is identical to the parameter `from_tz`. However, the result type is still the TIMESTAMP data type.
- The optional fourth parameter (string) specifies options how problematic input data due to time shifts should be handled. The following alternatives exist:

```
'INVALID SHIFT AMBIGUOUS ST'
'INVALID SHIFT AMBIGUOUS DST'
'INVALID SHIFT AMBIGUOUS NULLIFY'
'INVALID SHIFT AMBIGUOUS REJECT'
'INVALID ADJUST AMBIGUOUS ST'
'INVALID ADJUST AMBIGUOUS DST'
'INVALID ADJUST AMBIGUOUS NULLIFY'
'INVALID ADJUST AMBIGUOUS REJECT'
'INVALID NULLIFY AMBIGUOUS ST'
'INVALID NULLIFY AMBIGUOUS DST'
'INVALID NULLIFY AMBIGUOUS NULLIFY'
'INVALID NULLIFY AMBIGUOUS REJECT'
'INVALID REJECT AMBIGUOUS ST'
'INVALID REJECT AMBIGUOUS DST'
'INVALID REJECT AMBIGUOUS NULLIFY'
'INVALID REJECT AMBIGUOUS REJECT'
'ENSURE REVERSIBILITY'
```

Details about the options can be found in section [Date/Time data types](#) in [Section 2.3, “Data types”](#). The last option is a special option to ensure the reversibility of the conversion. An exception is thrown if the input data is invalid or ambiguous, and if the result timestamp would be ambiguous (which means it couldn't be converted back without information loss).

When omitting the fourth parameter, the default behavior is defined by the session value `TIME_ZONE_BEHAVIOR` (see [ALTER SESSION](#)).

Example(s)

```
SELECT CONVERT_TZ(TIMESTAMP '2012-05-10 12:00:00',
                  'UTC',
                  'Europe/Berlin') CONVERT_TZ;
```

```

CONVERT_TZ
-----
2012-05-10 14:00:00

SELECT CONVERT_TZ(TIMESTAMP '2012-03-25 02:30:00',
                  'Europe/Berlin',
                  'UTC',
                  'INVALID REJECT AMBIGUOUS REJECT') CONVERT_TZ;

Error: [22034] data exception -
       unable to convert timestamp between timezones: invalid timestamp

```

CORR

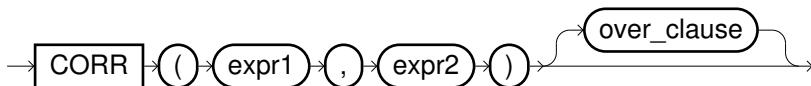
Purpose

Returns the coefficient of correlation of a set of number pairs (a type of relation measure). This equates to the following formula:

$$\text{CORR(expr1, expr2)} = \frac{\text{COVAR_POP(expr1, expr2)}}{\text{STDDEV_POP(expr1) \cdot STDDEV_POP(expr2)}}$$

Syntax

`corr ::=`



Note(s)

- If either `expr1` or `expr2` is the value `NULL`, then the corresponding number pair is not considered for the computation.
- See also [Section 2.9.3, “Analytical functions”](#) for the `OVER()` clause and analytical functions in general.

Example(s)

```

SELECT industry, CORR(age, salary) CORR
FROM staff GROUP BY industry;

INDUSTRY      CORR
-----
Finance      0.966045513268967
IT            0.453263345203583

```

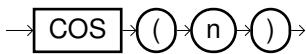
COS

Purpose

Returns the cosine of number n.

Syntax

`cos::=`



Example(s)

```
SELECT COS(PI()/3);  
COS(PI()/3)  
-----  
      0.5
```

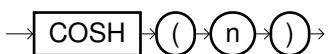
COSH

Purpose

Returns the hyperbolic cosine of number `n`.

Syntax

`cosh::=`



Example(s)

```
SELECT COSH(1);  
COSH(1)  
-----  
1.543080634815244
```

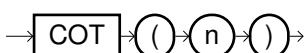
COT

Purpose

Returns the cotangent of number `n`.

Syntax

`cot::=`



Example(s)

```
SELECT COT(1);

COT(1)
-----
0.642092615934331
```

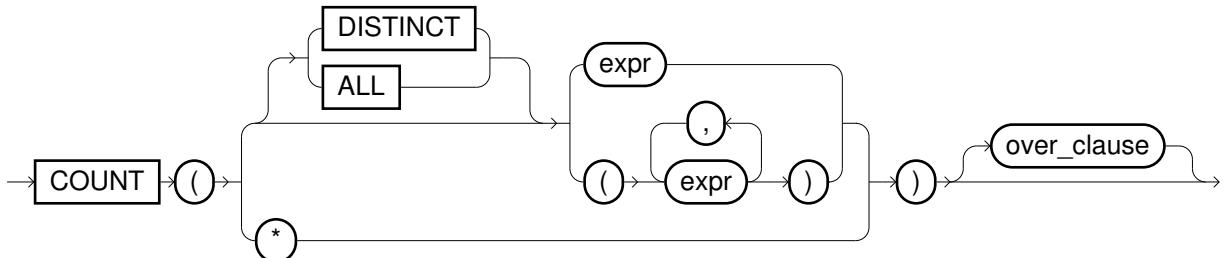
COUNT

Purpose

Returns the number of rows in the result set of a SQL query.

Syntax

count ::=



Note(s)

- If * is specified, all of the rows are counted.
- If an expression is specified, the NULL values are not counted. If you define tuples, then those tuples are not counted which consist only of NULL values.
- Duplicate entries are only counted once with DISTINCT; with ALL, all occurrences are counted.
- The default value is ALL if neither ALL nor DISTINCT is specified.
- A fast, but not exact alternative to COUNT(DISTINCT) is the function [APPROXIMATE_COUNT_DISTINCT](#).

Example(s)

```
SELECT COUNT(*) CNT_ALL FROM staff;

CNT_ALL
-----
10

SELECT COUNT(DISTINCT salary) CNT_DIST FROM staff;

CNT_DIST
-----
8
```

COVAR_POP

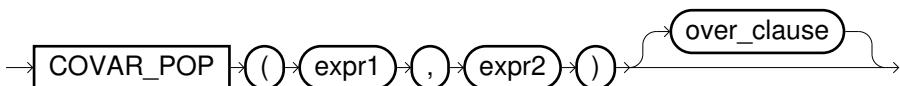
Purpose

Returns the population covariance of a set of number pairs (a type of relation measure). This equates to the following formula:

$$\text{COVAR_POP(expr1, expr2)} = \frac{\sum_{i=1}^n (\text{expr1}_i - \overline{\text{expr1}})(\text{expr2}_i - \overline{\text{expr1}})}{n}$$

Syntax

`covar_pop ::=`



Note(s)

- If either `expr1` or `expr2` is the value `NULL`, then the corresponding number pair is not considered for the computation.
- See also [Section 2.9.3, “Analytical functions”](#) for the `OVER()` clause and analytical functions in general.

Example(s)

```

SELECT industry, COVAR_POP(age, salary) COVAR_POP
FROM staff GROUP BY industry;

INDUSTRY      COVAR_POP
-----
Finance          209360
IT                31280
  
```

COVAR_SAMP

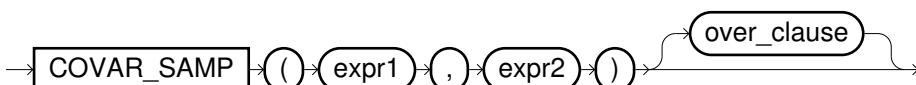
Purpose

Returns the sample covariance of a set of number pairs (a type of relation measure). This equates to the following formula:

$$\text{COVAR_SAMP(expr1, expr2)} = \frac{\sum_{i=1}^n (\text{expr1}_i - \overline{\text{expr1}})(\text{expr2}_i - \overline{\text{expr1}})}{n - 1}$$

Syntax

`covar_samp ::=`



Note(s)

- If either `expr1` or `expr2` is the value `NULL`, then the corresponding number pair is not considered for the computation.
- See also [Section 2.9.3, “Analytical functions”](#) for the `OVER()` clause and analytical functions in general.

Example(s)

```
SELECT industry, COVAR_SAMP(age, salary) COVAR_SAMP
FROM staff GROUP BY industry;

INDUSTRY    COVAR_SAMP
-----
Finance        261700
IT            39100
```

CURDATE

Purpose

Returns the current date by evaluating `TO_DATE(CURRENT_TIMESTAMP)`.

Syntax

`curdate ::=`

→ **CURDATE** → () →)

Note(s)

- This function is an alias for [CURRENT_DATE](#).

Example(s)

```
SELECT CURDATE( ) CURDATE;

CURDATE
-----
1999-12-31
```

CURRENT_DATE

Purpose

Returns the current date by evaluating `TO_DATE(CURRENT_TIMESTAMP)`.

Syntax

`current_date ::=`

→ **CURRENT_DATE** →

Note(s)

- See also function [CURDATE](#).

Example(s)

```
SELECT CURRENT_DATE;  
  
CURRENT_DATE  
-----  
1999-12-31
```

CURRENT_SCHEMA

Purpose

Returns the schema currently open. If a schema is not open, the NULL value is the result.

Syntax

current_schema::=

→ **CURRENT_SCHEMA** →

Example(s)

```
SELECT CURRENT_SCHEMA;  
  
CURRENT_SCHEMA  
-----  
MY_SCHEMA
```

CURRENT_SESSION

Purpose

Returns the id of the current session. This id is also referenced e.g. in system table [EXA_ALL_SESSIONS](#).

Syntax

current_session::=

→ **CURRENT_SESSION** →

Example(s)

```
SELECT CURRENT_SESSION;

CURRENT_SESSION
-----
7501910697805018352
```

CURRENT_STATEMENT

Purpose

Returns the current id of the statements which is serially numbered within the current session.

Syntax

current_statement::=

→ **CURRENT_STATEMENT** →

Example(s)

```
SELECT CURRENT_STATEMENT;

CURRENT_STATEMENT
-----
26
```

CURRENT_TIMESTAMP

Purpose

Returns the current timestamp, interpreted in the current session time zone.

Syntax

current_timestamp::=

→ **CURRENT_TIMESTAMP** →

Note(s)

- The return value is of data type TIMESTAMP WITH LOCAL TIME ZONE.
- The function **NOW** is an alias for CURRENT_TIMESTAMP.
- Other functions for the current moment:
 - **LOCALTIMESTAMP**
 - **SYSTIMESTAMP**

Example(s)

```
SELECT CURRENT_TIMESTAMP;  
  
CURRENT_TIMESTAMP  
-----  
1999-12-31 23:59:59
```

CURRENT_USER

Purpose

Returns the current user.

Syntax

current_user::=

→ **CURRENT_USER** →

Example(s)

```
SELECT CURRENT_USER;  
  
CURRENT_USER  
-----  
SYS
```

DATE_TRUNC

Purpose

PostgreSQL compatible function to round down date and timestamp values.

Syntax

date_trunc::=

→ **DATE_TRUNC** → (→ **format** → , → **datetime** →) →

Note(s)

- As format you can use one of the following elements: 'microseconds', 'milliseconds', 'second', 'minute', 'hour', 'day', 'week', 'month', 'quarter', 'year', 'decade', 'century', 'millennium'
- The first day of a week (for format element 'week') is defined via the parameter NLS_FIRST_DAY_OF_WEEK (see [ALTER SESSION](#) and [ALTER SYSTEM](#)).
- A similar functionality provides the Oracle compatible function [TRUNC\[ATE\] \(datetime\)](#).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT DATE_TRUNC( 'month' , DATE '2006-12-31' ) DATE_TRUNC;  
  
DATE_TRUNC  
-----  
2006-12-01  
  
SELECT DATE_TRUNC( 'minute' , TIMESTAMP '2006-12-31 23:59:59' ) DATE_TRUNC;  
  
DATE_TRUNC  
-----  
2006-12-31 23:59:00.000000
```

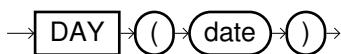
DAY

Purpose

Returns the day of a date.

Syntax

day::=



Note(s)

- This function can also be applied on strings, in contrast to function [EXTRACT](#).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT DAY(DATE '2010-10-20') ;  
  
DAY  
---  
20
```

DAYS_BETWEEN

Purpose

Returns the number of days between two date values.

Syntax

days_between::=



Note(s)

- If a timestamp is entered, only the date contained therein is applied for the computation.
- If the first date value is earlier than the second date value, the result is negative.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT DAYS_BETWEEN(DATE '1999-12-31',DATE '2000-01-01') DB1,
       DAYS_BETWEEN(TIMESTAMP '2000-01-01 12:00:00',
                     TIMESTAMP '1999-12-31 00:00:00') DB2;

DB1          DB2
-----
-1           1
```

DBTIMEZONE

Purpose

Returns the database time zone which is set system-wide in EXAoperation and represents the local time zone of the EXASOL servers.

Syntax

dbtimezone ::=

→ **DBTIMEZONE** →

Note(s)

- See also [SESSIONTIMEZONE](#).

Example(s)

```
SELECT DBTIMEZONE;

DBTIMEZONE
-----
EUROPE/BERLIN
```

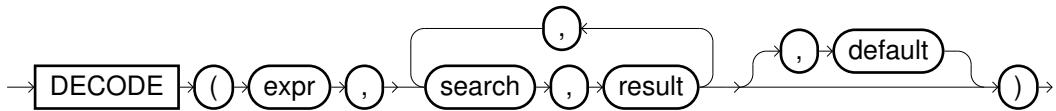
DECODE

Purpose

The decode function returns the `result` value for which the expression, `expr`, matches the expression, `search`. If no match is found, `NULL` or – if specified – the `default` value is returned.

Syntax

decode ::=



Note(s)

- Decode is similar to [CASE](#), but has slightly different functionality (to be compliant to other databases):
 - The expression `expr` can be directly compared with value `NULL` (e.g. `DECODE(my_column,NULL,0,my_column)`)
 - String comparisons are done "non-padded" (that's why `DECODE(my_column,'abc',TRUE,FALSE)` on a `CHAR(10)` column always returns `false`)
- Due to readability reasons we recommend to use [CASE](#).

Example(s)

```

SELECT DECODE('abc', 'xyz', 1, 'abc', 2, 3) DECODE;
DECODE
-----
2
  
```

DEGREES

Purpose

Returns the corresponding angle in degrees for an angle specified in radians.

Syntax

degrees ::=



Note(s)

- See also function [RADIAN](#).
- The input variable can be any number.
- The result range is $(-180;180]$.

Example(s)

```

SELECT DEGREES(PI());
DEGREES(PI())
-----
180
  
```

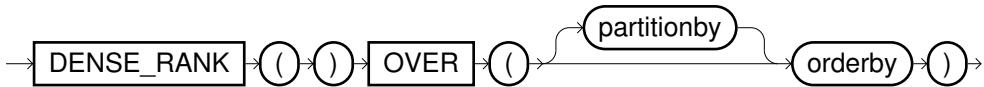
DENSE_RANK

Purpose

Returns the rank of a row within an ordered partition.

Syntax

dense_rank ::=



Note(s)

- DENSE_RANK can only be used as an analytical function (in combination with OVER(...), see also [Section 2.9.3, “Analytical functions”](#)).
- The OVER clause must contain an ORDER BY part and may not contain a window clause.
- The same value is returned for rows with equal ranking. However, the following values are not skipped - as is the case with [RANK](#).

Example(s)

```
SELECT name, salary, DENSE_RANK() OVER (ORDER BY salary DESC) dense_rank
FROM staff ORDER BY dense_rank;
```

NAME	SALARY	DENSE_RANK
Schmidt	81000	1
Müller	75000	2
Huber	48000	3
Schultze	37000	4
Schulze	37000	4
Meier	25000	5

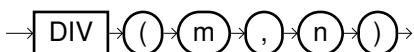
DIV

Purpose

Returns the integer quotient of m and n.

Syntax

div ::=



Example(s)

```
SELECT DIV(15,6) DIV;
```

```
DIV
---
2
```

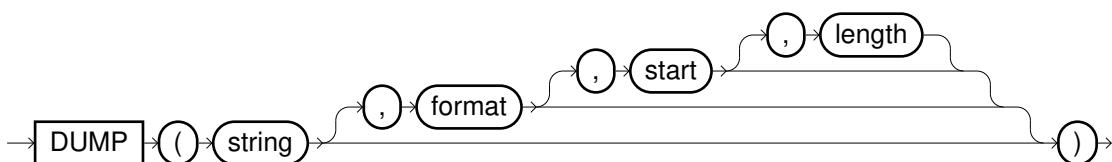
DUMP

Purpose

Returns the byte length and the character set of `string`, as well as the internal representation of the characters specified by startposition `start` and length `length`.

Syntax

`dump ::=`



Note(s)

- The argument `format` specifies the format of the return value. There are four valid format-values:
 - 8: Octal notation
 - 10: Decimal notation (Default)
 - 16: Hexadecimal notation
 - 17: ASCII characters are directly printed, multi-byte-characters are printed in hexadecimal format
- The argument `length` specifies the maximal number of selected characters beginning at startposition `start`. If `length=0` all possible characters are selected. For negative numbers the absolute value of `length` will be used.
- The argument `start` specifies the startposition of the character selection. If the character length of `string` is less than the absolute value of `start` the function returns NULL. For negative numbers the startposition is set to the absolute value of `start` counted from the right (Default=1).
- If the argument `string` is NULL the function returns the character string 'NULL'.

Example(s)

```

SELECT DUMP( '123abc' ) DUMP;

DUMP
-----
Len=6 CharacterSet=ASCII: 49,50,51,97,98,99

SELECT DUMP( 'üääö45' ,16 ) DUMP;

DUMP
-----
Len=8 CharacterSet=UTF8: c3,bc,c3,a4,c3,b6,34,35
  
```

EDIT_DISTANCE

Purpose

This function defines the distance between two strings, indicating how similar they are.

Syntax

edit_distance ::=

→ **EDIT_DISTANCE** → (→ string → , → string →) →

Note(s)

- To check the phonetic equivalence of strings you can use the functions [SOUNDEX](#) and [COLOGNE_PHONETIC](#).
- The number of changes is calculated which need to be done to convert one string into the other.
- The result is a number between 0 and the length of the wider string.

Example(s)

```
SELECT EDIT_DISTANCE('schmitt', 'Schmidt');

EDIT_DISTANCE('schmitt', 'Schmidt')
-----
2
```

EXP

Purpose

Returns the number e (Euler's number) to the power of n.

Syntax

exp ::=

→ **EXP** → (→ n →) →

Example(s)

```
SELECT EXP(1);

EXP(1)
-----
2.718281828459045
```

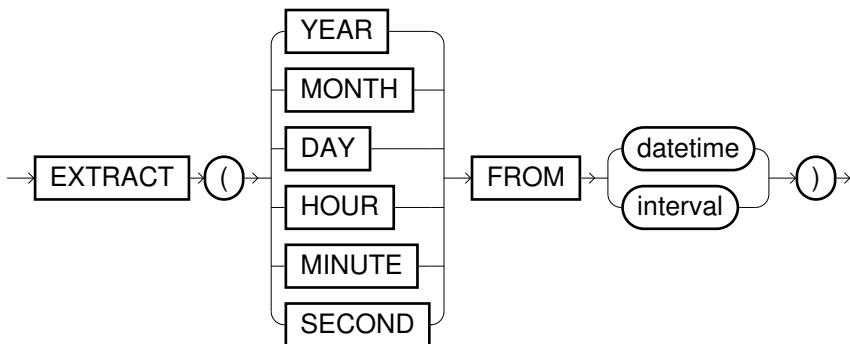
EXTRACT

Purpose

Extracts specific values from a timestamp, date or interval.

Syntax

extract ::=



Note(s)

- Valid parameters for the different data types:

DATE	YEAR, MONTH, DAY
TIMESTAMP	YEAR, MONTH, DAY, HOUR, MINUTE, SECOND
TIMESTAMP WITH LOCAL TIME ZONE	YEAR, MONTH, DAY, HOUR, MINUTE, SECOND
INTERVAL YEAR TO MONTH	YEAR, MONTH
INTERVAL DAY TO SECOND	DAY, HOUR, MINUTE, SECOND

- When extracting seconds, the milliseconds contained in the timestamp or interval are also extracted.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```

SELECT EXTRACT(SECOND FROM TIMESTAMP '2000-10-01 12:22:59.123') EXS,
       EXTRACT(MONTH FROM DATE '2000-10-01') EXM,
       EXTRACT(HOUR FROM INTERVAL '1 23:59:30.123' DAY TO SECOND) EXH;

EXS      EXM  EXH
-----  -----
 59.123   10   23
  
```

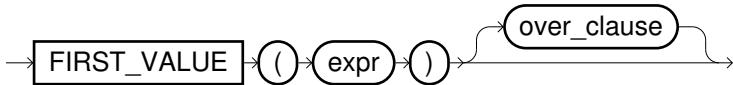
FIRST_VALUE

Purpose

Returns the first row in a group.

Syntax

first_value ::=



Note(s)

- Due to the fact that the rows in EXASOL are distributed across the cluster, FIRST_VALUE is non-deterministic as an aggregate function. Accordingly, FIRST_VALUE serves primarily as a help function in the event that only the same elements are contained within a group.
- The same applies when being used as an analytical function (see also [Section 2.9.3, “Analytical functions”](#).) if the OVER clause does not include an ORDER BY part.

Example(s)

```

SELECT name,
       hire_date,
       FIRST_VALUE(hire_date) OVER (ORDER BY hire_date) FIRST_VAL
FROM staff;

NAME      HIRE_DATE      FIRST_VAL
-----
mueller   '2013-05-01'  '2013-05-01'
schmidt   '2013-08-01'  '2013-05-01'
meier     '2013-10-01'  '2013-05-01'
  
```

FLOOR

Purpose

Returns the largest whole number that is smaller or equal to n.

Syntax

floor ::=



Example(s)

```

SELECT FLOOR(4.567) FLOOR;

FLOOR
-----
 4
  
```

FROM_POSIX_TIME

Purpose

Posix time (also known as Unix time) is a system for describing points in time, defined as the number of seconds elapsed since midnight of January 1, 1970 (UTC). By using this function you can convert the Posix Time (that means a numerical value) to a timestamp.

Syntax

from_posix_time ::=



Note(s)

- `FROM_POSIX_TIME(<number>)` is equivalent to the function call `ADD_SECONDS('1970-01-01 00:00:00', <number>)` if the session time zone is set to UTC.
- If you pass a negative number, the function will return timestamps before January 1, 1970 (UTC).
- Via the function [POSIX_TIME](#) you can compute the Posix time, that means you can convert a datetime value into a numeric value.

Example(s)

```
ALTER SESSION SET TIME_ZONE='UTC';SELECT FROM_POSIX_TIME(1) FPT1,
    FROM_POSIX_TIME(1234567890) FPT2;
FPT1          FPT2
-----
1970-01-01 00:00:01.000000 2009-02-13 23:31:30.000000
```

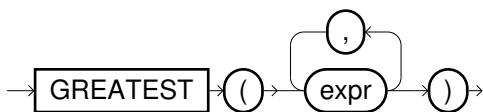
GREATEST

Purpose

Returns the largest of the specified expressions.

Syntax

greatest ::=



Note(s)

- The data type `BOOLEAN` is not supported.

Example(s)

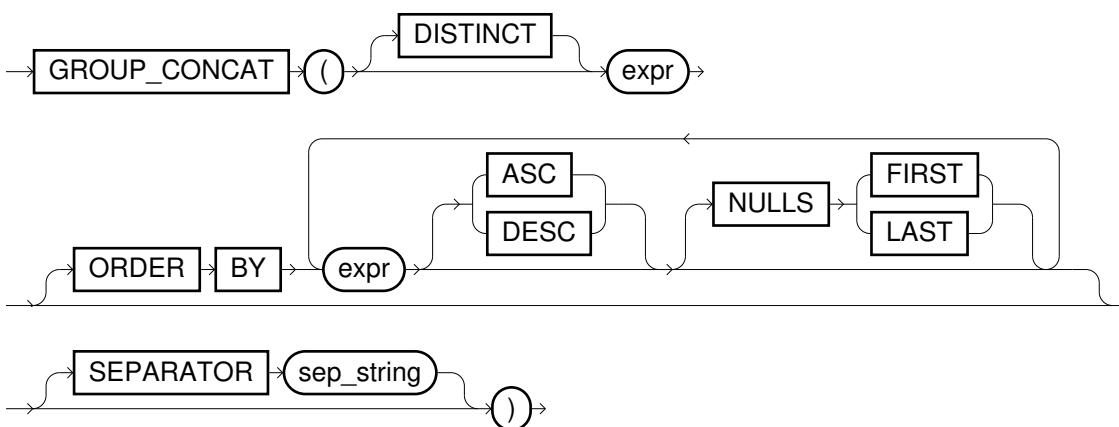
```
SELECT GREATEST(1,5,3) GREATEST;
GREATEST
-----
5
```

GROUP_CONCAT**Purpose**

Concatenates the values of `concat_expr` for each group.

Syntax

`group_concat ::=`

**Note(s)**

- If you specify `DISTINCT`, duplicate strings are eliminated, if they would be in series.
- When using `ORDER BY`, the rows within a group are sorted before the aggregation (concatenation). In default case, the options `ASC NULLS LAST` are used (ascending, `NULL` values at the end).
- If the `ORDER BY` option is omitted and the `DISTINCT` option is specified, then the rows of a group are implicitly sorted by `concat_expr` to ensure that the result is deterministic.
- By using `SEPARATOR sep_string`, you can define the delimiter between the concatenated elements. In default case the delimiter `' , '` is used. By using the empty string `('')` you can also omit the delimiter completely.
- The resulting data type is the maximal string type (`VARCHAR(2000000)`).

Example(s)

```
SELECT department, GROUP_CONCAT(name ORDER BY name)
FROM staff GROUP BY department;

DEPARTMENT  GROUP_CONCAT
-----  -----
sales      carsten,joe,thomas
marketing   alex,monica
```

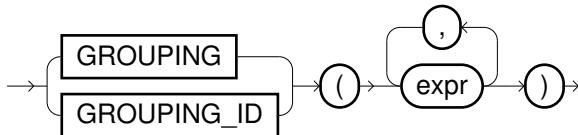
GROUPING[_ID]

Purpose

By the use of this function you can distinguish between regular result rows and superaggregate rows which are created in case of GROUPING SETS, CUBE or ROLLUP clauses.

Syntax

grouping ::=



Note(s)

- Each argument must be similar to an expression within the GROUP BY clause.
- In case of a single argument the result value is 0 if the corresponding grouping considers this expression, otherwise 1 (superaggregation).
- In case of multiple arguments the result value is a number whose binary representation is similar to GROUPING(arg₁),GROUPING(arg₂),..., GROUPING(arg_n). E.g. the following is true:

$$\text{GROUPING}(a,b,c) = 4 \times \text{GROUPING}(a) + 2 \times \text{GROUPING}(b) + 1 \times \text{GROUPING}(c)$$

- Details to GROUPING SETS, CUBE and ROLLUP can be found in the notes of the command [SELECT](#) in [Section 2.2.4, “Query language \(DQL\)”](#).

Example(s)

SELECT SUM(volume) revenue, y, m, DECODE(GROUPING(y,m),1,'yearly',3,'total',NULL) superaggregate FROM sales GROUP BY ROLLUP(y,m) ORDER BY y,revenue;			
REVENUE	Y	M	SUPERAGGREGATE
1725.90	2010	December	
1725.90	2010		yearly
735.88	2011	April	
752.46	2011	February	
842.32	2011	March	
931.18	2011	January	
3261.84	2011		yearly
4987.74			total

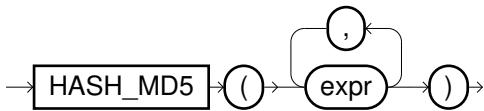
HASH_MD5

Purpose

Computes a hash value by using the MD5 algorithm (128 Bit).

Syntax

hash_md5 ::=



Note(s)

- Return values have data type CHAR (32) and contain hex characters.
- The data types of the input parameters are significant. That is why HASH_MD5(123) is different to HASH_MD5('123').
- Multiple input expressions are concatenated (in their internal byte representation) before the hash value is computed. Please note that generally, HASH_MD5(c1, c2) is not similar to HASH_MD5(c1 || c2).
- The function returns NULL if all input expressions are NULL.

Example(s)

```

SELECT HASH_MD5( 'abc' );
HASH_MD5( 'abc' )
-----
900150983cd24fb0d6963f7d28e17f72

```

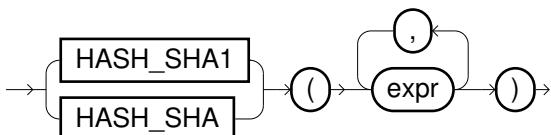
HASH_SHA[1]

Purpose

Computes a hash value by using the SHA1 algorithm (160 Bit).

Syntax

hash_sha1 ::=



Note(s)

- Return values have data type CHAR (40) and contain hex characters.
- The data types of the input parameters are significant. That is why HASH_SHA1(123) is different to HASH_SHA1('123').
- Multiple input expressions are concatenated (in their internal byte representation) before the hash value is computed. Please note that generally, HASH_SHA1(c1, c2) is not similar to HASH_SHA1(c1 || c2).
- The function returns NULL if all input expressions are NULL.
- HASH_SHA() is an alias for HASH_SHA1().

Example(s)

```
SELECT HASH_SHA1( 'abc' );  
  
HASH_SHA1( 'abc' )  
-----  
a9993e364706816aba3e25717850c26c9cd0d89d
```

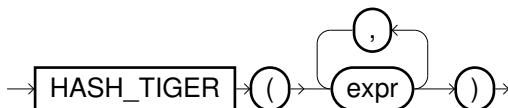
HASH_TIGER

Purpose

Computes a hash value by using the tiger algorithm (192 Bit).

Syntax

hash_tiger ::=



Note(s)

- Return values have data type CHAR (48) and contain hex characters.
- The data types of the input parameters are significant. That is why HASH_TIGER(123) is different to HASH_TIGER('123').
- Multiple input expressions are concatenated (in their internal byte representation) before the hash value is computed. Please note that generally, HASH_TIGER(c1,c2) is not similar to HASH_TIGER(c1||c2).
- The function returns NULL if all input expressions are NULL.

Example(s)

```
SELECT HASH_TIGER( 'abc' );  
  
HASH_TIGER( 'abc' )  
-----  
2aab1484e8c158f2bfb8c5ff41b57a525129131c957b5f93
```

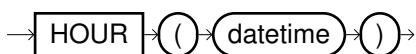
HOUR

Purpose

Returns the hours of a timestamp.

Syntax

hour ::=



Note(s)

- This function can also be applied on strings, in contrast to function [EXTRACT](#).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT HOUR(TIMESTAMP '2010-10-20 11:59:40.123') ;  
  
HOU  
---  
11
```

HOURS_BETWEEN

Purpose

Returns the number of hours between timestamp `timestamp1` and timestamp `timestamp2`.

Syntax

`hours_between`::=

→ **HOURS_BETWEEN** → (→ **datetime1** → , → **datetime2** →) →

Note(s)

- If timestamp `timestamp1` is earlier than timestamp `timestamp2`, then the result is negative.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated internally within UTC.

Example(s)

```
SELECT HOURS_BETWEEN(TIMESTAMP '2000-01-01 12:00:00',  
                      TIMESTAMP '2000-01-01 11:01:05.1') HB;  
  
HB  
-----  
0.981916666667
```

INSERT

Purpose

Replaces the substring of `string`, with length `length` beginning at `position`, with string `new_string`.

Syntax

`insert`::=



Note(s)

- The first character of string has position 1. If the variable position is 0 or outside the string, then the string isn't changed. If it is negative, then the function counts backwards from the end.
- If length=0, then new_string is just inserted and nothing is replaced.
- If position+length>length(string) or if length<0, then the string is replaced beginning from position.
- If one of the parameters is NULL, then NULL is returned.

Example(s)

```

SELECT INSERT( 'abc' ,2,2,'xxx') ,
       INSERT( 'abcdef' ,3,2,'CD');

INSERT( 'abc' ,2,2,'xxx')  INSERT( 'abcdef' ,3,2,'CD')
-----
axxx                      abCDef
  
```

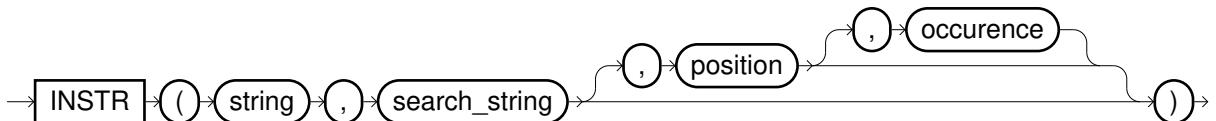
INSTR

Purpose

Returns the position in string at which search_string appears. If this is not contained, the value 0 is returned.

Syntax

instr ::=



Note(s)

- The optional parameter position defines from which position the search shall begin (the first character has position 1). If the value is negative, EXASOL counts *and* searches backwards from the end (e.g. INSTR(string, 'abc', -3) searches backwards from the third last letter).
- The optional positive number occurrence defines which occurrence shall be searched for.
- INSTR(string,search_string) is similar to INSTR(string,search_string,1,1).
- The functions POSITION and LOCATE are similar.

Example(s)

```

SELECT INSTR( 'abcabcabc' , 'cab' ) INSTR1,
       INSTR( 'user1,user2,user3,user4,user5' , 'user' , -1 , 2 ) INSTR2;

INSTR1      INSTR2
  
```

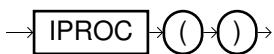
IPROC

Purpose

Returns the local node number within the cluster. By that, you can visualize which rows are stored on which nodes.

Syntax

iproc ::=



Note(s)

- The result value is an integer between 0 and [NPROC](#)-1.
- Only active database nodes are count, but no reserve nodes.
- In this context, please also note functions [NPROC](#) and [VALUE2PROC](#).

Example(s)

```

SELECT c1, IPROC() IPROC FROM t ORDER BY c1;

C1 IPROC
-- -----
1 0
2 1
3 2
4 3
5 0
6 1

SELECT IPROC() IPROC, COUNT(*) CNT FROM t GROUP BY 1;

IPROC CNT
-- --
0      2
1      2
2      1
3      1
  
```

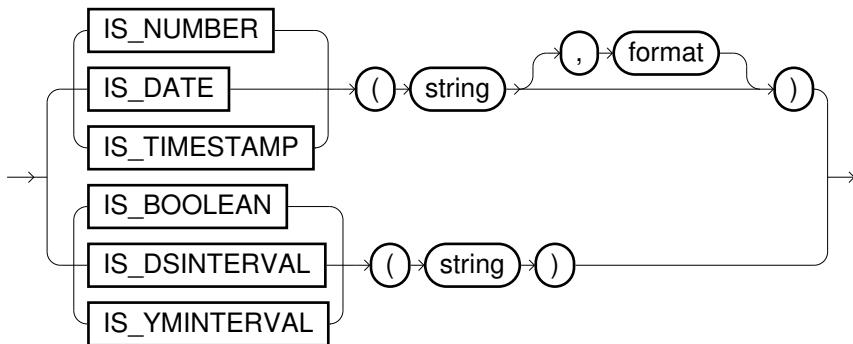
IS_*

Purpose

Returns TRUE if `string` can be converted to a certain data type. If e.g. `IS_NUMBER` returns TRUE, you can convert the string via [TO_NUMBER](#).

Syntax

is_datatype ::=



Note(s)

- If one of the arguments is NULL, then NULL is returned.
 - If a format is specified, then the rules of the corresponding TO functions apply ([TO_NUMBER](#), [TO_DATE](#), [TO_TIMESTAMP](#), [TO_DSINTERVAL](#), [TO_YMINTERVAL](#)). See also [Section 2.6.2](#), “Numeric format models”.

Example(s)

```
SELECT IS_BOOLEAN('xyz') IS_BOOLEAN,
       IS_NUMBER('+12.34') IS_NUMBER,
       IS_DATE('12.13.2011', 'DD.MM.YYYY') IS_DATE;

IS_BOOLEAN IS_NUMBER IS_DATE
-----
FALSE      TRUE      FALSE
```

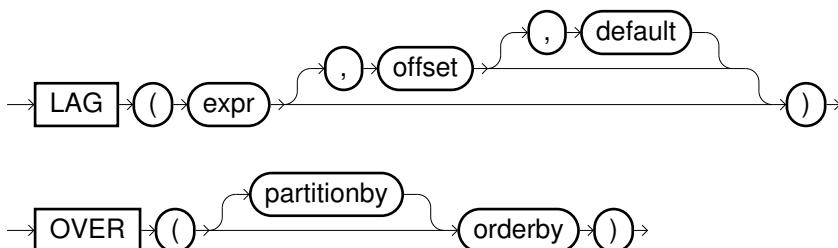
LAG

Purpose

By using LAG in an analytical function, you are able to access previous rows within a partition. The expression `expr` is evaluated on that row which is located exactly `offset` rows prior to the current row.

Syntax

lag::=



Note(s)

- LAG can only be used as an analytical function (in combination with OVER(...), see also [Section 2.9.3, "Analytical functions"](#)).
- The OVER clause must contain an ORDER BY part and may not contain a window clause.
- If the ORDER BY part doesn't define an unique sort order, the result is non-deterministic.
- If the access is beyond the scope of the current partition, LAG returns the value of parameter default or NULL if default was not specified.
- If you omit the parameter offset, value 1 is used. Negative values are not allowed for offset. Rows with offset NULL are handled as if they were beyond the scope of the current partition.
- To access following rows you can use the function [LEAD](#).

Example(s)

```

SELECT airplane, maintenance_date, last_maintenance,
       DAYS_BETWEEN(maintenance_date, last_maintenance) unmaintained_time
FROM
(
    SELECT airplane, maintenance_date,
           LAG(maintenance_date,1)
              OVER (PARTITION BY airplane
                     ORDER BY maintenance_date) last_maintenance
    FROM maintenance
    WHERE TRUNC(maintenance_date, 'MM')='2008-06-01'
)
ORDER BY airplane, maintenance_date;

AIRPLANE          MAINTENANC LAST_MAINT UNMAINTAIN
-----          -----      -----      -----
Banana airlines - jet 1 2008-06-03
Banana airlines - jet 1 2008-06-29  2008-06-03          26
Safe travel - jet 1   2008-06-02
Safe travel - jet 1   2008-06-09  2008-06-02          7
Safe travel - jet 1   2008-06-16  2008-06-09          7
Safe travel - jet 1   2008-06-23  2008-06-16          7
Safe travel - jet 1   2008-06-30  2008-06-23          7

```

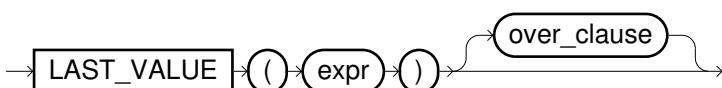
LAST_VALUE

Purpose

Returns the last row in a group.

Syntax

last_value::=



Note(s)

- Due to the fact that the rows in EXASOL are distributed across the cluster, LAST_VALUE is non-deterministic as an aggregate function. Accordingly, LAST_VALUE serves primarily as a help function in the event that only the same elements are contained within a group.
- The same applies when being used as an analytical function (see also [Section 2.9.3, “Analytical functions”](#).) if the OVER clause does not include an ORDER BY part.

Example(s)

```
SELECT name,
       hire_date,
       LAST_VALUE(hire_date) OVER (ORDER BY hire_date
                                     ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) LAST_VAL
  FROM staff;

NAME      HIRE_DATE      LAST_VAL
-----  -----
mueller   '2013-05-01'  '2013-10-01'
schmidt   '2013-08-01'  '2013-10-01'
meier     '2013-10-01'  '2013-10-01'
```

LCASE**Purpose**

Converts the specified string into lower case letters.

Syntax

lcase ::=

→ **LCASE** → (→ **string** →) →

Note(s)

- LCASE is an alias for [LOWER](#).

Example(s)

```
SELECT LCASE( 'AbCdEf' ) LCASE;

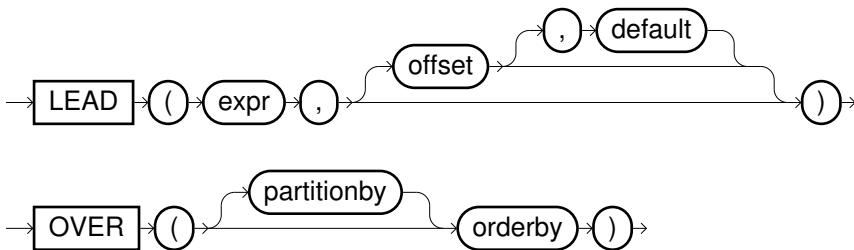
LCASE
-----
abcdef
```

LEAD**Purpose**

By using LEAD in an analytical function, you are able to access following rows within a partition. The expression `expr` is evaluated on that row which is located exactly `offset` rows beyond the current row.

Syntax

lead::=



Note(s)

- LEAD can only be used as an analytical function (in combination with OVER(...), see also [Section 2.9.3, “Analytical functions”](#)).
- The OVER clause must contain an ORDER BY part and may not contain a window clause.
- If the ORDER BY part doesn't define an unique sort order, the result is non-deterministic.
- If the access is beyond the scope of the current partition, LEAD returns the value of parameter default or NULL if default was not specified.
- If you omit the parameter offset, value 1 is used. Negative values are not allowed for offset. Rows with offset NULL are handled as if they were beyond the scope of the current partition.
- To access previous rows you can use the function [LAG](#).

Example(s)

```

SELECT company, fiscal_year, market_value, expected_growth,
       CAST( (next_market_value - market_value) / market_value
             AS DECIMAL(5,2) ) real_growth,
       next_market_value - market_value net_increase
FROM
  ( SELECT company, fiscal_year, market_value, expected_growth,
          LEAD( market_value, 1 )
          OVER ( PARTITION BY company
                  ORDER BY fiscal_year ) next_market_value
    FROM yearly_market_value )
ORDER BY company, fiscal_year;
  
```

COMPANY	FISCAL_YEAR	MARKET_VALUE	EXPECTE	REAL_GR	NET_INCREASE
Bankers R US	2004	20.00	0.15	0.25	5.00
Bankers R US	2005	25.00	0.25	0.20	5.00
Bankers R US	2006	30.00	0.17	0.33	10.00
Bankers R US	2007	40.00	0.22	-0.55	-22.00
Bankers R US	2008	18.00	-0.25		
Service Inc.	2004	102.00	0.05	0.08	8.00
Service Inc.	2005	110.00	0.07	0.09	10.00
Service Inc.	2006	120.00	0.05	0.04	5.00
Service Inc.	2007	125.00	0.03	-0.02	-3.00
Service Inc.	2008	122.00	-0.02		

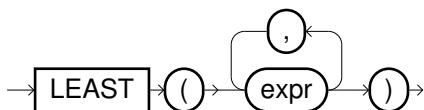
LEAST

Purpose

Returns the smallest of the specified expressions.

Syntax

least ::=



Note(s)

- The data type BOOLEAN is not supported.

Example(s)

```
SELECT LEAST(3,1,5) LEAST;
LEAST
-----
1
```

LEFT

Purpose

Returns the left-aligned substring of `string` with length `length`.

Syntax

left ::=



Note(s)

- If either `length` or `string` is NULL, then NULL is returned.
- Also if `length` is greater than the length of `string`, the original `string` is returned. If `length` is negative or 0, then NULL is returned.
- See also functions [SUBSTR\[ING\]](#), [MID](#) and [RIGHT](#).

Example(s)

```
SELECT LEFT('abcdef',3) LEFT_SUBSTR;
LEFT_SUBSTR
```

```
-----  
abc
```

LENGTH

Purpose

Returns the length of a string in characters.

Syntax

length::=

→ LENGTH → (→ string →) →

Example(s)

```
SELECT LENGTH( 'abc' ) LENGTH;  
  
LENGTH  
-----  
3
```

LEVEL

Purpose

Returns for CONNECT BY queries the level of a node within the tree. Details can be found in the description of the [SELECT statement](#) in [Section 2.2.4, “Query language \(DQL\)”](#).

Syntax

level::=

→ LEVEL →

Example(s)

```
SELECT last_name,  
       LEVEL,  
       SYS_CONNECT_BY_PATH(last_name, '/') "PATH"  
  FROM employees  
 CONNECT BY PRIOR employee_id = manager_id  
 START WITH last_name = 'Clark';  
  
LAST_NAME LEVEL PATH  
----- ----- -----  
Clark      1 /Clark  
Smith      2 /Clark/Smith  
Brown      3 /Clark/Smith/Brown
```

LN

Purpose

Returns the natural logarithm of number n. The function LN(n) is equivalent to LOG(EXP(1),n).

Syntax

ln ::=



Note(s)

- The specified number n must be greater than 0.

Example(s)

```
SELECT LN(100) LN;  
  
LN  
-----  
4.6051701859881
```

LOCALTIMESTAMP

Purpose

Returns the current timestamp, interpreted in the current session time zone.

Syntax

localtimestamp ::=



Note(s)

- The return value is of data type TIMESTAMP.
- Other functions for the current moment:
 - CURRENT_TIMESTAMP or NOW
 - SYSTIMESTAMP

Example(s)

```
SELECT LOCALTIMESTAMP;  
  
LOCALTIMESTAMP  
-----  
2000-12-31 23:59:59.000
```

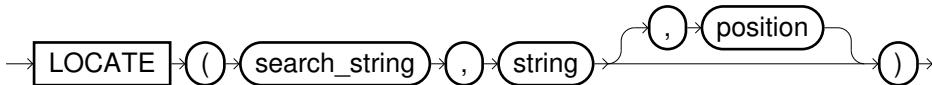
LOCATE

Purpose

Returns the position in `string` at which `search_string` appears. If this is not contained, the value 0 is returned.

Syntax

`locate ::=`



Note(s)

- The optional parameter `position` defines from which position the search shall begin (starting with 1). If the value is negative, EXASOL counts *and* searches backwards from the end (e.g. `LOCATE('abc', string, -3)` searches backwards from the third last letter).
- `LOCATE(search_string, string)` is similar to `LOCATE(search_string, string, 1)`.
- The functions `POSITION` and `INSTR` are similar.

Example(s)

```
SELECT LOCATE('cab', 'abcababcabc') LOCATE1,
       LOCATE('user', 'user1,user2,user3,user4,user5', -1) LOCATE2;
LOCATE1      LOCATE2
-----      -----
            3          25
```

LOG

Purpose

Returns the logarithm of `n` with base `base`.

Syntax

`log ::=`



Note(s)

- The number `base` must be positive and must not be 1.
- The number `n` must be positive.

Example(s)

```
SELECT LOG(2,1024);  
  
LOG(2,1024)  
-----  
10
```

LOG10

Purpose

Returns the logarithm of n with base 10.

Syntax

log10::=



Note(s)

- The number n must be positive.

Example(s)

```
SELECT LOG10(10000) LOG10;  
  
LOG10  
-----  
4
```

LOG2

Purpose

Returns the logarithm of n with base 2.

Syntax

log2::=



Note(s)

- The number n must be positive.

Example(s)

```
SELECT LOG2(1024) LOG2;  
  
LOG2  
-----  
      10
```

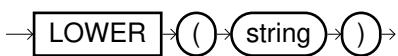
LOWER

Purpose

Converts the specified string into lower case letters.

Syntax

lower ::=



Note(s)

- LOWER is an alias for [LCASE](#).

Example(s)

```
SELECT LOWER( 'AbCdEf' );  
  
LOWER( 'AbCdEf' )  
-----  
abcdef
```

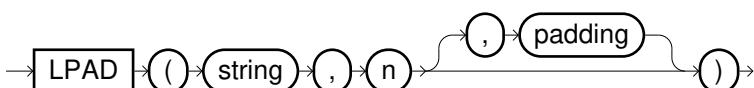
LPAD

Purpose

Returns a string of length n, which is string, filled from the left with expression padding.

Syntax

lpad ::=



Note(s)

- If the variable padding is not specified, spaces are used.
- Even if n is greater than 2,000,000, the result string is truncated to 2,000,000 characters.
- For filling a string from the right, please refer to function [RPAD](#).

Example(s)

```
SELECT LPAD( 'abc' , 5 , 'X' ) ;

LPAD( 'abc' , 5 , 'X' )
-----
XXabc
```

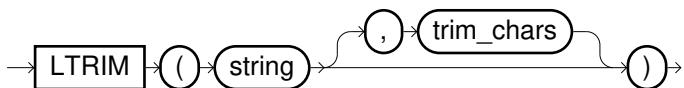
LTRIM

Purpose

LTRIM deletes all of the characters specified in the expression `trim_chars` from the left border of `string`.

Syntax

`ltrim`::=



Note(s)

- If parameter `trim_chars` is not specified, spaces are deleted.

Example(s)

```
SELECT LTRIM( 'ab cdef' , ' ab' ) ;

LTRIM( 'ab cdef' , ' ab' )
-----
cdef
```

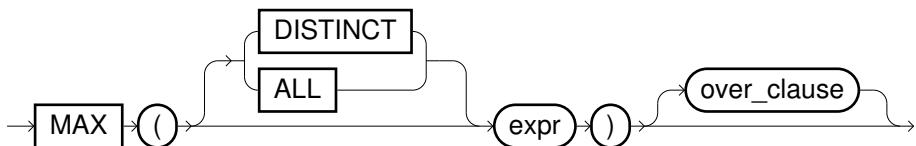
MAX

Purpose

Returns the maximum.

Syntax

`max`::=



Note(s)

- It is irrelevant for the result whether ALL or DISTINCT is specified.

Example(s)

```
SELECT MAX(age) MAX FROM staff;

MAX
-----
57
```

MEDIAN**Purpose**

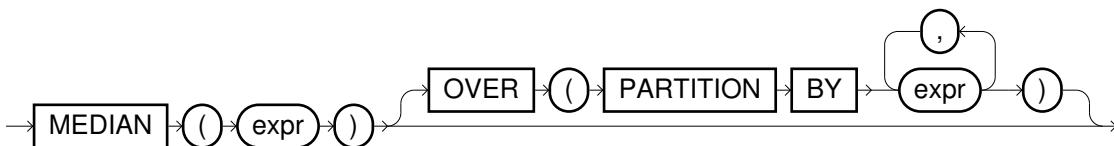
MEDIAN is an inverse distribution function. In contrast to the average function (see [AVG](#)) the median function returns the middle value or an interpolated value which would be the middle value once the elements are sorted (NULL values are ignored).

The following formula is evaluated:

$$\text{MEDIAN(expr)} = \begin{cases} \text{expr}_{(n+1)/2} & \text{for } n \text{ odd} \\ \frac{\text{expr}_{n/2} + \text{expr}_{n/2+1}}{2} & \text{for } n \text{ even} \end{cases}$$

Syntax

median::=

**Note(s)**

- MEDIAN(<expr>) is an alias for PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY <expr>).
- See also the inverse distribution functions [PERCENTILE_CONT](#) and [PERCENTILE_DISC](#).

Example(s)

```
COUNT
-----
50
100
200
900

SELECT MEDIAN(count) MEDIAN FROM sales;

MEDIAN
```

150

MID

Purpose

Returns a substring of length `length` from position `position` out of the string `string`.

Syntax

`mid`::=



Note(s)

- If `length` is not specified, all of the characters to the end of the string are used.
- The first character of a string has position 1. If `position` is negative, counting begins at the end of the string.
- See also functions `RIGHT` and `LEFT`.
- `MID` is an alias for `SUBSTR[ING]`.

Example(s)

```
SELECT MID( 'abcdef' , 2 , 3 ) MID;  
MID  
---  
bcd
```

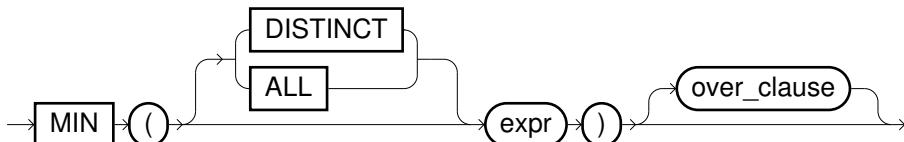
MIN

Purpose

Returns the minimum.

Syntax

`min`::=



Note(s)

- It is irrelevant for the result whether `ALL` or `DISTINCT` is specified.

Example(s)

```
SELECT MIN(age) MIN FROM staff;  
  
MIN  
-----  
25
```

MINUTE

Purpose

Returns the minutes of a timestamp.

Syntax

minute ::=

→ **MINUTE** → (→ **datetime** →) →

Note(s)

- This function can also be applied on strings, in contrast to function [EXTRACT](#).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT MINUTE(TIMESTAMP '2010-10-20 11:59:40.123');  
  
MIN  
---  
59
```

MINUTES_BETWEEN

Purpose

Returns the number of minutes between two timestamps `timestamp1` and `timestamp2`.

Syntax

minutes_between ::=

→ **MINUTES_BETWEEN** → (→ **datetime1** → , → **datetime2** →) →

Note(s)

- If timestamp `timestamp1` is earlier than timestamp `timestamp2`, then the result is negative.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated internally within UTC.

Example(s)

```
SELECT MINUTES_BETWEEN(TIMESTAMP '2000-01-01 12:01:00',
                        TIMESTAMP '2000-01-01 12:00:02') MINUTES;

MINUTES
-----
0.966666666667
```

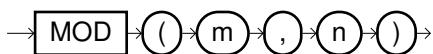
MOD

Purpose

Delivers the remainder of the division of m by n.

Syntax

mod ::=



Example(s)

```
SELECT MOD(15,6) MODULO;

MODULO
-----
3
```

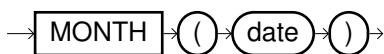
MONTH

Purpose

Returns the month of a date.

Syntax

month ::=



Note(s)

- This function can also be applied on strings, in contrast to function [EXTRACT](#).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT MONTH(DATE '2010-10-20');

MON
---
10
```

MONTHS_BETWEEN

Purpose

Returns the number of months between two date values.

Syntax

months_between ::=

→ **MONTHS_BETWEEN** → (→ **datetime1** → , → **datetime2** →) →

Note(s)

- If a timestamp is entered, only the date contained therein is applied for the computation.
- If the days are identical or both are the last day of a month, the result is an integer.
- If the first date value is earlier than the second date value, the result is negative.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT MONTHS_BETWEEN(DATE '2000-01-01', DATE '1999-12-15') MB1,
       MONTHS_BETWEEN(TIMESTAMP '2000-01-01 12:00:00',
                      TIMESTAMP '1999-06-01 00:00:00') MB2;

MB1          MB2
-----
0.548387096774194    7
```

NOW

Purpose

Returns the current timestamp, interpreted in the current session time zone.

Syntax

now ::=

→ **NOW** → (→) →

Note(s)

- The return value is of data type TIMESTAMP.



Please note that the result data type will be changed from TIMESTAMP to TIMESTAMP WITH LOCAL TIME ZONE in the next major version. To avoid any influence on the existing processes in a minor version, we delayed that change. The impact will be rather small, since the values will still be the same (current timestamp interpreted in the session time zone), only the data type will differ.

- This function is an alias for [CURRENT_TIMESTAMP](#).
- Other functions for the current moment:
 - [LOCALTIMESTAMP](#)
 - [SYSTIMESTAMP](#)

Example(s)

```
SELECT NOW( ) NOW;  
  
NOW  
-----  
1999-12-31 23:59:59
```

NPROC

Purpose

This function returns the number of database nodes in the cluster.

Syntax

nproc ::=

→ **NPROC** → () → () →

Note(s)

- Only active database nodes are counted, but no reserve nodes.
- In this context, please also note functions [IPROC](#) and [VALUE2PROC](#).

Example(s)

```
SELECT NPROC( ) NPROC;  
  
NPROC  
-----  
4
```

NULLIF

Purpose

Returns the value NULL if two expressions are identical. Otherwise, the first expression is returned.

Syntax

nullif ::=

→ **NULLIF** → (→ expr1 → , → expr2 →) →

Note(s)

- The NULLIF function is equivalent to the CASE expression CASE WHEN expr1=expr2 THEN NULL ELSE expr1 END

Example(s)

```
SELECT NULLIF(1,2) NULLIF1, NULLIF(1,1) NULLIF2;  
NULLIF1 NULLIF2  
----- -----  
          1
```

NULLIFZERO

Purpose

Returns the value NULL if number has value 0. Otherwise, number is returned.

Syntax

nullifzero ::=

→ **NULLIFZERO** → (→ number →) →

Note(s)

- The NULLIFZERO function is equivalent to the CASE expression CASE WHEN number=0 THEN NULL ELSE number END.
- See also [ZEROIFNULL](#).

Example(s)

```
SELECT NULLIFZERO(0) NIZ1, NULLIFZERO(1) NIZ2;  
NIZ1 NIZ2  
----- -----  
          1
```

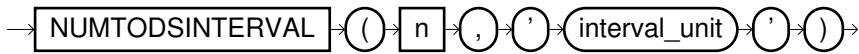
NUMTODSINTERVAL

Purpose

Converts a numerical value n into an interval of type INTERVAL DAY TO SECOND.

Syntax

numtodsinterval ::=



Note(s)

- The parameter `interval_unit` can be either DAY, HOUR, MINUTE or SECOND.
- See also [NUMTOYMINTERVAL](#), [TO_DSINTERVAL](#) and [TO_YMINTERVAL](#).

Example(s)

```
SELECT NUMTODSINTERVAL( 3.2, 'HOUR' ) NUMTODSINTERVAL;  
-----  
NUMTODSINTERVAL  
+000000000 03:12:00.000000000
```

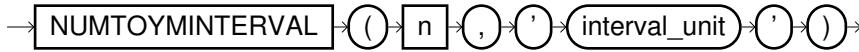
NUMTOYMINTERVAL

Purpose

Converts a numerical value n into an interval of type INTERVAL YEAR TO MONTH.

Syntax

numtoyminterval ::=



Note(s)

- The parameter `interval_unit` is either YEAR or MONTH.
- See also [NUMTODSINTERVAL](#), [TO_DSINTERVAL](#) and [TO_YMINTERVAL](#).

Example(s)

```
SELECT NUMTOYMINTERVAL( 3.5, 'YEAR' ) NUMTOYMINTERVAL;  
-----  
NUMTOYMINTERVAL  
+000000003-06
```

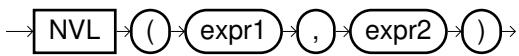
NVL

Purpose

Replaces NULL values with the expression, expr2.

Syntax

nvl ::=



Note(s)

- If expr1 is NULL, expr2 is returned, otherwise expr1.
- The abbreviation NVL stands for "Null Value".
- See also [ZEROIFNULL](#).

Example(s)

```
SELECT NVL(NULL, 'abc') NVL_1, NVL('xyz', 'abc') NVL_2;
----- -----
NVL_1  NVL_2
abc      xyz
```

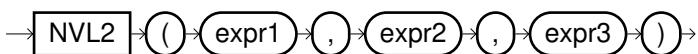
NVL2

Purpose

Replaces NULL values with expr3, otherwise it uses expr2.

Syntax

nvl2 ::=



Note(s)

- If expr1 is NULL, expr3 is returned, otherwise expr2.
- The abbreviation NVL stands for "Null Value".
- If there is no legal type conversion between expr2 and expr3, the function returns an error.
- See also [NVL](#).

Example(s)

```
SELECT NVL2(NULL, 2, 3) NVL_1,
       NVL2(1, 2, 3) NVL_2;
```

NVL_1	NVL_2
---	---
3	2

OCTET_LENGTH

Purpose

Returns the octet length of a string. If only ASCII characters are used, then this function is equivalent to [CHARACTER_LENGTH](#) and [LENGTH](#).

Syntax

octet_length ::=

→ **OCTET_LENGTH** → (→ **string** →) →

Example(s)

```
SELECT OCTET_LENGTH( 'abcd' ) OCT_LENGTH;

OCT_LENGTH
-----
4

SELECT OCTET_LENGTH( 'äöü' ) OCT_LENGTH;

OCT_LENGTH
-----
6
```

PERCENTILE_CONT

Purpose

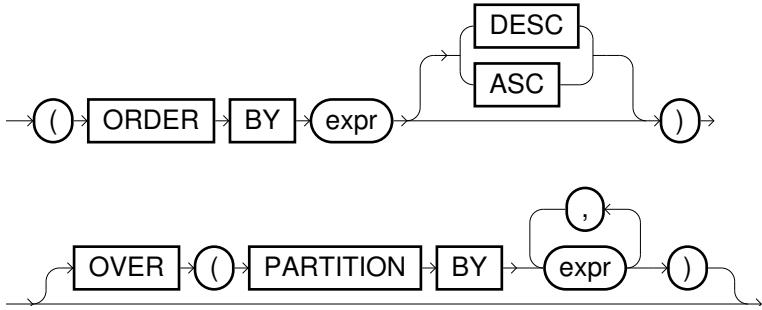
PERCENTILE_CONT is an inverse distribution function and expects as input parameter a percentile value and a sorting specification which defines the rank of each element within a group. The functions returns the percentile of this sort order (e.g. in case of percentile 0.7 and 100 values, the 70th value is returned).

If the percentile cannot be assigned exactly to an element, then the linear interpolation between the two nearest values is returned (e.g. in case of percentile 0.71 and 10 values, the interpolation between the 7th and 8th value).

Syntax

percentile_cont ::=

→ **PERCENTILE_CONT** → (→ **expr** →) → **WITHIN** → **GROUP** →



Note(s)

- NULL values are ignored for the computation.
- The specified percentile value must be constant (between 0 and 1).
- If neither DESC nor ASC is specified, then ASC is used.
- See also the inverse distribution functions [PERCENTILE_DISC](#) and [MEDIAN](#).

Example(s)

COUNT	REGION	COUNTRY
100	EUROPE	NETHERLANDS
500	EUROPE	UK
600	EUROPE	FRANCE
700	EUROPE	GERMANY
900	ASIA	CHINA
300	ASIA	KOREA
700	ASIA	JAPAN

SELECT region,
PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY count),
PERCENTILE_CONT(0.9) WITHIN GROUP (ORDER BY count)
FROM sales
GROUP BY region;

REGION	PERCENTILE_CONT(0.5)	PERCENTILE_CONT(0.9)
EUROPE	550	670
ASIA	700	860

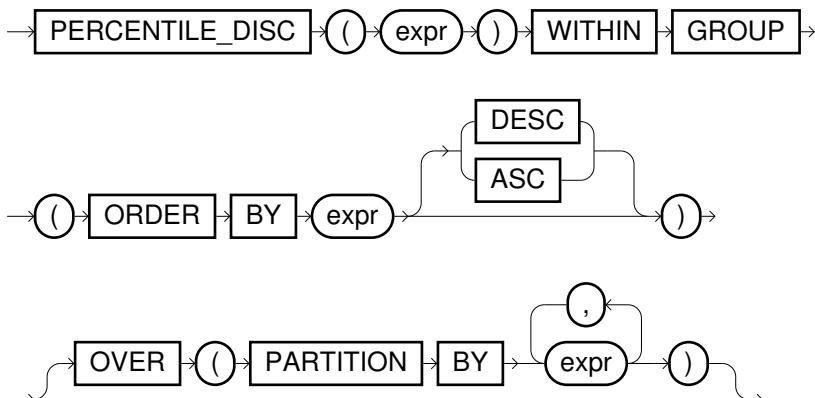
PERCENTILE_DISC

Purpose

PERCENTILE_DISC is an inverse distribution function and returns the value from the group set which has the smallest cumulative distribution value (corresponding to the given sort order) which is larger than or equal to the specified percentile value. NULL values are ignored for the calculation.

Syntax

percentile_disc::=

**Note(s)**

- The specified percentile value must be constant (between 0 and 1).
- PERCENTILE_DISC(0) always returns the first value of the group set.
- If neither DESC nor ASC is specified, then ASC is used.
- See also the inverse distribution functions [PERCENTILE_CONT](#) and [MEDIAN](#).

Example(s)

COUNT	REGION	COUNTRY
100	EUROPE	NETHERLANDS
500	EUROPE	UK
600	EUROPE	FRANCE
700	EUROPE	GERMANY
900	ASIA	CHINA
300	ASIA	KOREA
700	ASIA	JAPAN

SELECT region,
PERCENTILE_DISC(0.5) WITHIN GROUP (ORDER BY count),
PERCENTILE_DISC(0.7) WITHIN GROUP (ORDER BY count)
FROM sales
GROUP BY region;

REGION	PERCENTILE_DISC(0.5)	PERCENTILE_DISC(0.7)
EUROPE	500	600
ASIA	700	900

PI**Purpose**

Returns the value of the mathematical constant π (pi).

Syntax

pi::=



Example(s)

```
SELECT PI( );
PI
-----
3.141592653589793
```

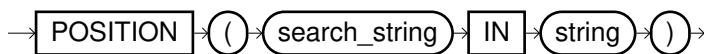
POSITION

Purpose

Returns the position in the string, `string`, at which the string, `search_string`, first appears. If this is not contained, the value 0 is returned.

Syntax

`position ::=`



Note(s)

- If one of the arguments is `NULL`, then `NULL` is returned.
- The functions `INSTR` and `LOCATE` are similar to this function.

Example(s)

```
SELECT POSITION('cab' IN 'abcababc') POS;
POS
-----
3
```

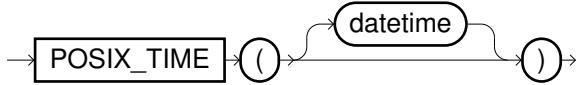
POSIX_TIME

Purpose

Posix time (also known as Unix time) is a system for describing points in time, defined as the number of seconds elapsed since midnight of January 1, 1970 (UTC). By using this function you can convert a datetime value to a numerical value.

Syntax

`posix_time ::=`

**Note(s)**

- `POSIX_TIME(<datetime>)` is equivalent to the function call `SECONDS_BETWEEN(<datetime>, '1970-01-01 00:00:00')` if the session time zone is set to UTC.
- If you omit the parameter, the Posix time refers at the moment (that means `CURRENT_TIMESTAMP`).
- For datetime values before January 1, 1970 (UTC), this function will return negative numbers.
- Via the function `FROM_POSIX_TIME` you can convert a numerical value into a datetime value.

Example(s)

```

ALTER SESSION SET TIME_ZONE='UTC';
SELECT POSIX_TIME('1970-01-01 00:00:01') PT1,
       POSIX_TIME('2009-02-13 23:31:30') PT2;

PT1          PT2
-----
1.000      1234567890.000
  
```

POWER**Purpose**

Returns the power of two numbers.

Syntax

power::=

**Example(s)**

```

SELECT POWER(2,10) POWER;

POWER
-----
1024
  
```

RADIANS**Purpose**

Converts the number n from degrees to radians.

Syntax

radians ::=



Note(s)

- See also function [DEGREES](#).

Example(s)

```
SELECT RADIANS(180) RADIANS;  
  
RADIANS  
-----  
3.141592653589793
```

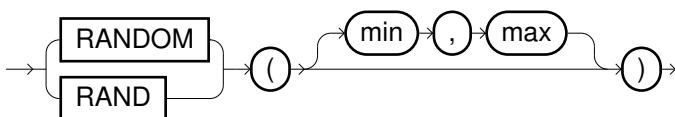
RAND[OM]

Purpose

Generates a random number.

Syntax

random ::=



Note(s)

- The result is always a DOUBLE.
- When specifying the optional parameters, a random number within the interval [min,max] is created. Without any parameters this interval is [0,1].

Example(s)

```
SELECT RANDOM() RANDOM_1, RANDOM(5,20) RANDOM_2;  
  
RANDOM_1           RANDOM_2  
----- -----  
0.379277567626116 12.7548096816858
```

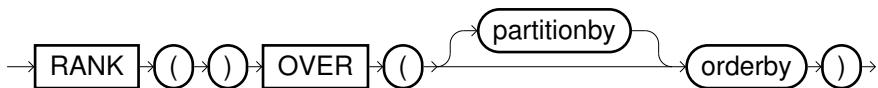
RANK

Purpose

Returns the rank of a row in an ordered partition.

Syntax

rank ::=



Note(s)

- RANK can only be used as an analytical function (in combination with OVER(...), see also [Section 2.9.3, "Analytical functions"](#)).
- The OVER clause must contain an ORDER BY part and may not contain a window clause.
- The same value is returned for rows with equal ranking. Therefore, afterwards a lot of values are omitted (as opposed to [DENSE_RANK](#)).

Example(s)

```
SELECT name, salary, RANK() OVER (ORDER BY Salary DESC) RANK
FROM staff ORDER BY rank;
```

NAME	SALARY	RANK
Schmidt	81000	1
Müller	75000	2
Huber	48000	3
Schultze	37000	4
Schulze	37000	4
Meier	25000	6

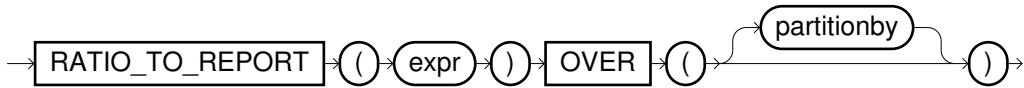
RATIO_TO_REPORT

Purpose

Computes the ratio of a value to the overall sum.

Syntax

ratio_to_report ::=



Note(s)

- The OVER clause may not include neither an ORDER BY nor a window clause (see also [Section 2.9.3, “Analytical functions”](#)).

Example(s)

```
SELECT name, salary, RATIO_TO_REPORT(salary) OVER () RATIO_TO_REPORT
FROM staff ORDER BY RATIO_TO_REPORT;
```

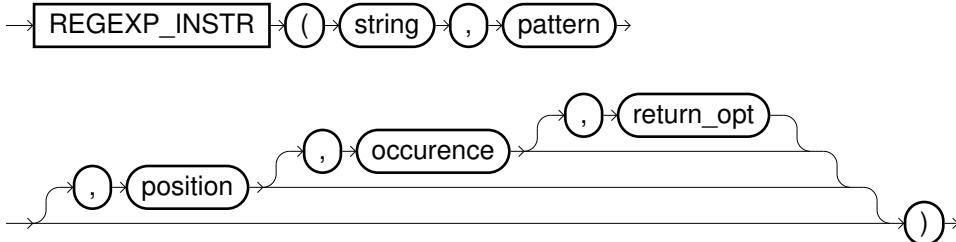
NAME	SALARY	RATIO_TO_REPORT
Meier	25000	0.082508250825083
Schultze	37000	0.122112211221122
Schulze	37000	0.122112211221122
Huber	48000	0.158415841584158
Müller	75000	0.247524752475248
Schmidt	81000	0.267326732673267

REGEXP_INSTR**Purpose**

Searches the regular expression pattern in string. If this is not contained, the value 0 is returned, otherwise the corresponding position of the match (see notes for details).

Syntax

regexp_instr::=

**Note(s)**

- Details and examples for regular expressions can be found in [Section 2.1.3, “Regular expressions”](#).
- The optional parameter position defines from which position the search shall begin (starting with 1).
- The optional positive number occurrence defines which occurrence shall be searched for. Please note that the search of the second occurrence begins at the first character after the first occurrence.
- The optional parameter return_opt defines the result of the function in case of a match:

0 (default) Function returns the beginning position of the match (counting starts from 1)

1 Function returns the end position of the match (character following the occurrence, counting starts from 1)

- REGEXP_INSTR(string, pattern) is similar to REGEXP_INSTR(string, pattern, 1, 1).
- See also functions [INSTR](#), [REGEXP_REPLACE](#) and [REGEXP_SUBSTR](#) and the predicate [\[NOT\] REGEXP LIKE](#).

Example(s)

```
SELECT REGEXP_INSTR('Phone: +497003927877678',
                     '\+?\d+'
                   ) REGEXP_INSTR1,
       REGEXP_INSTR('From: my_mail@yahoo.com - To: SERVICE@EXASOL.COM',
                     '(?i)[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}',
                     1,
                     2
                   ) REGEXP_INSTR2;

REGEXP_INSTR1  REGEXP_INSTR2
-----
8              31
```

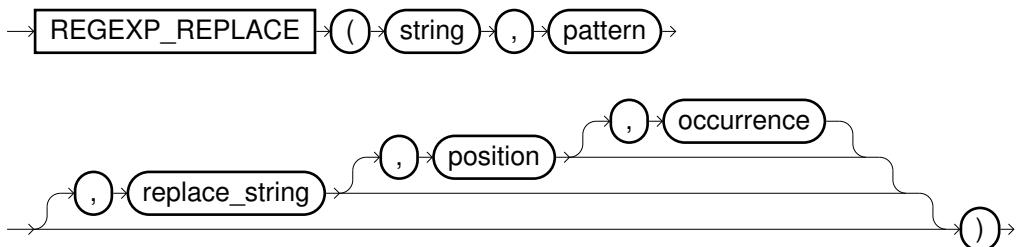
REGEXP_REPLACE

Purpose

Replaces occurrences of `pattern` in `string` by `replace_string`.

Syntax

`regexp_replace`::=



Note(s)

- Details and examples for regular expressions can be found in [Section 2.1.3, “Regular expressions”](#).
- If `pattern` is `NULL`, `string` is returned.
- If `replace_string` is omitted or `NULL`, the matches of `pattern` are deleted in the result.
- In `replace_string` you can use captures via `\1`, `\2`, ..., `\9` or `\g<name>` which are defined by `pattern`.
- The optional parameter `position` defines from which position the search shall begin (starting with 1).
- The optional positive number `occurrence` defines which occurrence shall be searched for. Please note that occurrences do not overlap. So the search of the second occurrence begins at the first character after the first occurrence. In case of 0 all occurrences are replaced (default). In case of a positive integer n the n-th occurrence will be replaced.
- See also functions `REPLACE`, `REGEXP_INSTR` and `REGEXP_SUBSTR` and the predicate `[NOT] REGEXP_LIKE`.

Example(s)

```
SELECT REGEXP_REPLACE(
    'From: my_mail@yahoo.com',
    '(?i)^From: ([a-z0-9._%+-]+)@([a-z0-9.-]+\.[a-z]{2,4}\$)',
    'Name: \1 - Domain: \2') REGEXP_REPLACE;
```

REGEXP_REPLACE

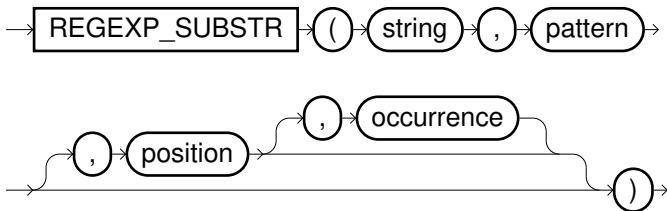
```
Name: my_mail - Domain: yahoo.com
```

REGEXP_SUBSTR**Purpose**

Returns a substring of the parameter `string`.

Syntax

`regexp_substring`::=

**Note(s)**

- Details and examples for regular expressions can be found in [Section 2.1.3, “Regular expressions”](#).
- Function `REGEXP_SUBSTR` is similar to function `REGEXP_INSTR`, but it returns the whole matching substring instead of returning the position of the match.
- The parameter `pattern` defines a regular expression to be searched for. If no match is found, `NULL` is returned. Otherwise the corresponding substring is returned.
- The optional parameter `position` defines from which position the search shall begin (starting with 1).
- The optional positive number `occurrence` defines which occurrence shall be searched for. Please note that the search of the second occurrence begins at the first character after the first occurrence.
- `REGEXP_SUBSTR(string, pattern)` is similar to `REGEXP_SUBSTR(string, pattern, 1, 1)`.
- See also functions `SUBSTR[ING]`, `REGEXP_INSTR` and `REGEXP_REPLACE` and the predicate `[NOT] REGEXP_LIKE`.

Example(s)

```

SELECT REGEXP_SUBSTR('My mail address is my_mail@yahoo.com',
                     '(?i)[a-z0-9._%+-]+@[a-z0-9.-]+\.[a-z]{2,4}') EMAIL
;

EMAIL
-----
my_mail@yahoo.com

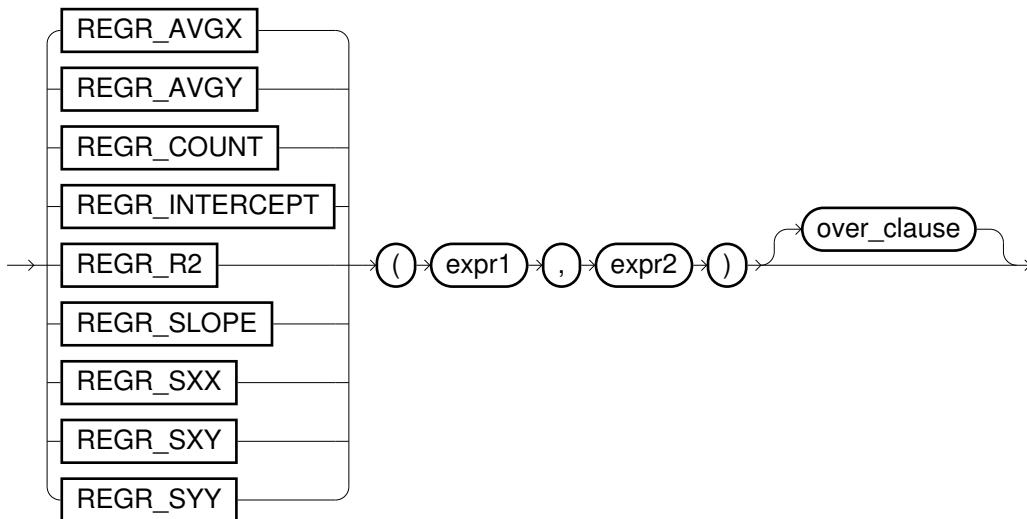
```

REGR_***Purpose**

With the help of the linear regression functions you can determine a least-square regression line.

Syntax

regr_functions ::=



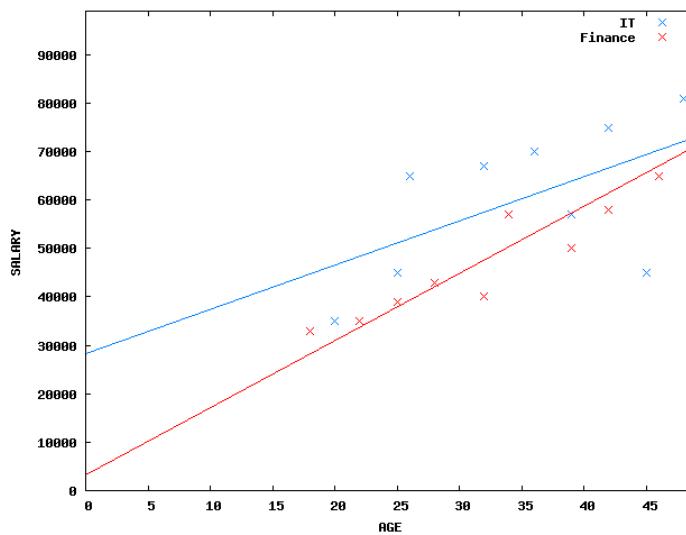
Note(s)

! *expr2* is interpreted as independent variable (x value), *expr1* as dependent variable (y value).

- If either *expr1* or *expr2* is NULL, then the corresponding number pair is not considered for the computation.
- Description for the regression functions:

Function	Description	Formula
REGR_SLOPE	Slope of the regression line	$\text{REGR_SLOPE(expr1, expr2)} = \frac{\text{COVAR_POP(expr1, expr2)}}{\text{VAR_POP(expr2)}}$
REGR_INTERCEPT	y-intercept of the regression line	$\text{REGR_INTERCEPT(expr1, expr2)} = \text{AVG(expr1)} - \text{REGR_SLOPE(expr1, expr2)} \cdot \text{AVG(expr2)}$
REGR_COUNT	Number of non-NULL number pairs	
REGR_R2	Coefficient of determination (goodness of fit).	$\text{REGR_R2(expr1, expr2)} = \begin{cases} \text{NULL} & \text{for } \text{VAR_POP(expr2)} = 0 \\ 1 & \text{for } \text{VAR_POP(expr2)} \neq 0, \text{VAR_POP(expr1)} = 0 \\ \text{CORR(expr1, expr2)}^2 & \text{else} \end{cases}$
REGR_AVGX	Average of the independent values (x values)	$\text{REGR_AVGX(expr1, expr2)} = \text{AVG(expr2)}$
REGR_AVGY	Average of the dependent values (y values)	$\text{REGR_AVGY(expr1, expr2)} = \text{AVG(expr1)}$
REGR_SXX	Auxiliary function	$\text{REGR_SXX(expr1, expr2)} = \text{REGR_COUNT(expr1, expr2)} \cdot \text{VAR_POP(expr2)}$
REGR_SXY	Auxiliary function	$\text{REGR_SXY(expr1, expr2)} = \text{REGR_COUNT(expr1, expr2)} \cdot \text{COVAR_POP(expr1, expr2)}$
REGR_SYY	Auxiliary function	$\text{REGR_SYY(expr1, expr2)} = \text{REGR_COUNT(expr1, expr2)} \cdot \text{VAR_POP(expr1)}$

- In the following example, two regression lines are determined. The crosses correspond to entries in the table staff (red=Finance, blue=IT). The two lines correspond to the determined regression lines. In the example you can see that the line for the finance sector shows more goodness (see REGR_R2 value), that means that the development of the salary is more dependent to the age.



Example(s)

```

SELECT industry,
       REGR_SLOPE(salary,age) AS REGR_SLOPE,
       REGR_INTERCEPT(salary,age) AS REGR_INTERCEPT,
       REGR_R2(salary,age) AS REGR_R2
  FROM staff GROUP BY industry;

INDUSTRY      REGR_SLOPE          REGR_INTERCEPT      REGR_R2
-----  -----
Finance      1385.778395127685  3314.563521743759  0.92187386620889
IT           913.8748888230062  28217.46219982212  0.325366059690104

```

REPEAT

Purpose

Returns the concatenation of n copies of a string.

Syntax

repeat::=

$\rightarrow \boxed{\text{REPEAT}} \rightarrow (\rightarrow \text{string} \rightarrow , \rightarrow n \rightarrow) \rightarrow$

Note(s)

- If one of the arguments is NULL or if n equals to 0, then NULL is returned.
- Parameter n must be a positive integer between 0 and 99999999.
- The result string may not contain more than 2000000 characters.
- If the input parameters is no string, it is automatically converted to a string.

- See also function [SPACE](#).

Example(s)

```
SELECT REPEAT( 'abc' , 3 ) ;  
  
REPEAT( 'abc' , 3 )  
-----  
abcabcabc
```

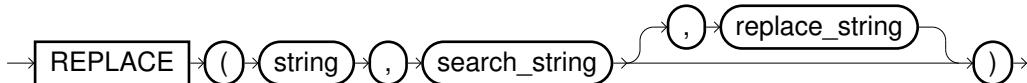
REPLACE

Purpose

Returns the string that emerges if in the `string` all occurrences of `search_string` are replaced by `replace_string`.

Syntax

`replace ::=`



Note(s)

- If `replace_string` is omitted or if it is NULL, all occurrences of `search_string` are deleted from the result.
- If `search_string` is NULL, `string` is returned.
- If the input parameters are not strings, they will be automatically converted to strings.
- The return type is always a string, even if all of the parameters possess another type.

Example(s)

```
SELECT REPLACE('Apple juice is great','Apple','Orange') REPLACE_1;  
  
REPLACE_1  
-----  
Orange juice is great  
  
SELECT REPLACE( '-TEST-Text-TEST-' , ' -TEST-' ) REPLACE_2;  
  
REPLACE_2  
-----  
Text
```

REVERSE

Purpose

Returns the reverse of a string value.

Syntax

reverse ::=



Note(s)

- If `string` is NULL, the function returns NULL.
- If the input parameter is not a string, it is automatically converted to strings.
- The return type is always a string which has the length of the input parameter.

Example(s)

```
SELECT REVERSE( 'abcde' ) REVERSE;
REVERSE
-----
edcba
```

RIGHT

Purpose

Returns the right-aligned substring of `string` with length `length`.

Syntax

right ::=



Note(s)

- If either `length` or `string` is NULL, then NULL is returned.
- Also if `length` is greater than the length of `string`, the original `string` is returned. If `length` is negative or 0, then NULL is returned.
- See also functions [SUBSTR\[ING\]](#), [MID](#) and [LEFT](#).

Example(s)

```
SELECT RIGHT( 'abcdef' , 3 ) RIGHT_SUBSTR;
RIGHT_SUBSTR
-----
def
```

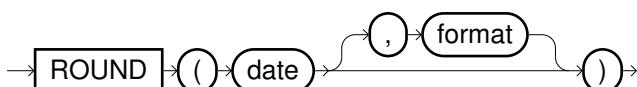
ROUND (datetime)

Purpose

Rounds a date or timestamp value to the specified unit.

Syntax

round (datetime)::=



Note(s)

- The following elements can be used as format:

CC, SCC	Century
YYYY, SYYY, YEAR, SYEAR,	Year
YYY, YY, Y	
IYYY, IYY, IY, I	Year in accordance with international standard, ISO 8601
Q	Quarter
MONTH, MON, MM, RM	Month
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD, DD, J	Day
DAY, DY, D	Starting day of the week. The first day of a week is defined via the parameter NLS_FIRST_DAY_OF_WEEK (see ALTER SESSION and ALTER SYSTEM).
HH, HH24, HH12	Hour
MI	Minute
SS	Seconds
- Rounding-up is as follows: years from July 1, months from the 16th of a month, days from 12 noon, hours from 30 minutes, minutes from 30 seconds and seconds from 500 milliseconds. Otherwise, values are rounded-down.
- If a format is not specified, the value is rounded to days.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```

SELECT ROUND(DATE '2006-12-31', 'YYYY') ROUND;

ROUND
-----
2007-01-01

SELECT ROUND(TIMESTAMP '2006-12-31 12:34:58', 'MI') ROUND;

ROUND
-----
2006-12-31 12:35:00.000
  
```

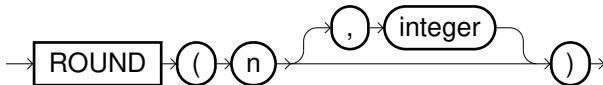
ROUND (number)

Purpose

Rounds number *n* to *integer* digits behind the decimal point (round to nearest, in case of a tie away of the zero).

Syntax

`round (number)::=`



Note(s)

- If the second argument is not specified, rounding is conducted to an integer.
- If the second argument is negative, rounding is conducted to *integer* digits in front of the decimal point.
- If the data type of *n* is `DECIMAL(p,s)` and the second argument is positive, then the result data type is `DECIMAL(p,integer)`.
- If the data type of *n* is `DOUBLE`, then the result data type is also `DOUBLE`. Due to the problem of representation for `DOUBLE` values, the result could contain numbers with more digits after the decimal point than you would expect because of the rounding. Therefore we recommend in this situation to always cast the result to an appropriate `DECIMAL` data type.

Example(s)

```

SELECT ROUND(123.456,2) ROUND;
-----+
123.46
  
```

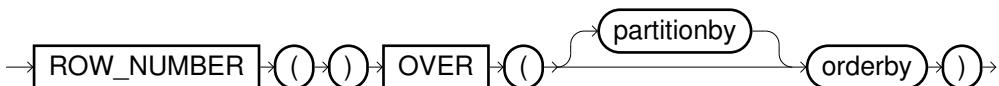
ROW_NUMBER

Purpose

Returns the number of a row in an ordered partition.

Syntax

`row_number::=`



Note(s)

- `ROW_NUMBER` can only be used as an analytical function (in combination with `OVER(...)`, see also [Section 2.9.3, “Analytical functions”](#)).
- The `OVER` clause must contain an `ORDER BY` part and may not contain a window clause.
- The value is non-deterministic with rows of equal ranking.

Example(s)

```
SELECT name, ROW_NUMBER() OVER (ORDER BY Salary DESC) ROW_NUMBER
FROM staff ORDER BY name;
```

NAME	ROW_NUMBER
Huber	3
Meier	6
Müller	2
Schmidt	1
Schultze	4
Schulze	5

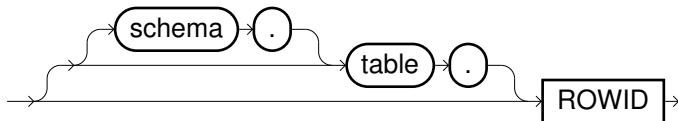
ROWID

Purpose

Every row of a base table in the database has a unique address, the so-called ROWID. Read access to this address can be obtained via the ROWID pseudo column (DECIMAL(36,0) data type).

Syntax

rowid ::=



Note(s)

The ROWID column can be used in, among others, the following SQL constructs:

- in all the column lists of SELECT statements
- in VIEW definitions (see also [CREATE VIEW](#)) in which case you have to define a column alias
- in conditions of [INSERT](#), [UPDATE](#), [MERGE](#) and [DELETE](#) statements

The ROWIDs of a table are managed by the DBMS. This ensures that the ROWIDs within a table are distinct - in contrast, it is quite acceptable for ROWIDs of different tables to be the same. Using DML statements such as [INSERT](#), [UPDATE](#), [DELETE](#), [TRUNCATE](#) or [MERGE](#), all the ROWIDs of the relevant tables are invalidated and reassigned by the DBMS. This compares to structural table changes such as adding a column, which leave the ROWIDs unchanged.

The ROWID pseudo column is only valid for base tables, not for views.

An example of using ROWIDs would be the targeted deletion of specific rows in a table, e.g. in order to restore the UNIQUE attribute.

Example(s)

```
SELECT ROWID, i FROM t;
```

ROWID	i
318815196395658560306020907325849600	1

```

318815196395658560306020907325849601          1
318815196395658560306302382302560256          2
318815196395658560306302382302560257          3

-- Restore the uniqueness of i
DELETE FROM t WHERE NOT EXISTS (
    SELECT r FROM (SELECT MIN(ROWID) r FROM t GROUP BY i) h
    WHERE t.ROWID=h.r);

CREATE VIEW v AS SELECT ROWID r, i FROM t;SELECT * FROM v;R
-----
318815196395658578752764981035401216          1
318815196395658578753046456012111872          2
318815196395658578753046456012111873          3

-- Error message, as only base tables have a ROWID column
SELECT ROWID FROM v;

Error: [42000] ROWID is invalid for non-material tables

```

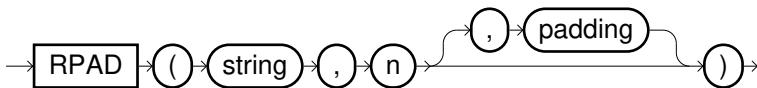
RPAD

Purpose

Returns a string of the length n, which is string, filled from the right with expression padding.

Syntax

rpad::=



Note(s)

- If the parameter padding is not specified, spaces are used.
- Even if n is greater than 2,000,000, the result string is truncated to 2,000,000 characters.
- For filling a string from the left, please refer to function [LPAD](#).

Example(s)

```

SELECT RPAD( 'abc' , 5 , 'X' );

RPAD( 'abc' , 5 , 'X' )
-----
abcXX

```

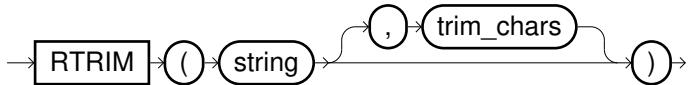
RTRIM

Purpose

RTRIM deletes all of the characters specified in the expression trim_chars from the right border of string.

Syntax

rtrim::=



Note(s)

- If parameter `trim_chars` is not specified, spaces are deleted.

Example(s)

```
SELECT RTRIM('abcdef','afe');

RTRIM('abcdef','afe')
-----
abcd
```

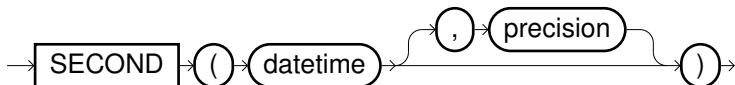
SECOND

Purpose

Returns the seconds of a timestamp.

Syntax

second::=



Note(s)

- The optional second parameter defines the number of digits behind the decimal point.
- This function can also be applied on strings, in contrast to function [EXTRACT](#).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT SECOND(TIMESTAMP '2010-10-20 11:59:40.123', 2);

SECOND
-----
40.12
```

SECONDS_BETWEEN

Purpose

Returns the number of seconds between two timestamps.

Syntax

seconds_between::=

→ **SECONDS_BETWEEN** → (→ **datetime1** → , → **datetime2** →) →

Note(s)

- If timestamp `timestamp1` is earlier than timestamp `timestamp2`, then the result is negative.
- Additionally, the result contains the difference in milliseconds.
- For data type `TIMESTAMP WITH LOCAL TIME ZONE` this function is calculated internally within UTC.

Example(s)

```
SELECT SECONDS_BETWEEN(TIMESTAMP '2000-01-01 12:01:02.345',
                      TIMESTAMP '2000-01-01 12:00:00') SB;
SB
-----
62.345
```

SESSIONTIMEZONE

Purpose

Returns the session time zone which was set via [ALTER SESSION](#).

Syntax

sessiontimezone::=

→ **SESSIONTIMEZONE** →

Note(s)

- See also [DBTIMEZONE](#).

Example(s)

```
SELECT SESSIONTIMEZONE;
SESSIONTIMEZONE
-----
EUROPE/BERLIN
```

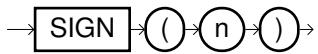
SIGN

Purpose

Returns the signum of number n as one of -1, 0, 1.

Syntax

sign ::=



Example(s)

```
SELECT SIGN(-123);  
  
SIGN(-123)  
-----  
-1
```

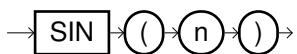
SIN

Purpose

Returns the sine of number n.

Syntax

sin ::=



Example(s)

```
SELECT SIN(PI() / 6);  
  
SIN(PI() / 6)  
-----  
0 . 5
```

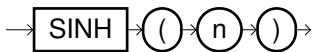
SINH

Purpose

Returns the hyperbolic sine of number n.

Syntax

sinh ::=



Example(s)

```
SELECT SINH(0) SINH;  
  
SINH  
-----  
          0
```

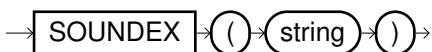
SOUNDEX

Purpose

SOUNDEX returns a phonetic representation of a string. You can use SOUNDEX to compare words which sounds similar, but are spelled different.

Syntax

soundex ::=



Note(s)

- For the computation of SOUNDEX the algorithm is used which is described in: *Donald Knuth, The Art of Computer Programming, Vol. 3.*
- The result is always a string with 4 characters (1 letter and 3 digits).
- This function is similar to [COLOGNE_PHONETIC](#) which is more appropriate for German words.

Example(s)

```
SELECT SOUNDEX('smythe'), SOUNDEX('Smith');  
  
SOUNDEX('smythe') SOUNDEX('Smith')  
----- -----  
S530           S530
```

SPACE

Purpose

SPACE creates a string consisting of n spaces.

Syntax

space ::=



Note(s)

- See also function [REPEAT](#).

Example(s)

```
SELECT 'x' || SPACE(5) || 'x' my_string;  
  
MY_STRING  
-----  
x      x
```

SQRT

Purpose

Returns the square root of number n.

Syntax

sqrt::=



Note(s)

- The number n must be ≥ 0 .

Example(s)

```
SELECT SQRT(2);  
  
SQRT(2)  
-----  
1.414213562373095
```

ST_*

Purpose

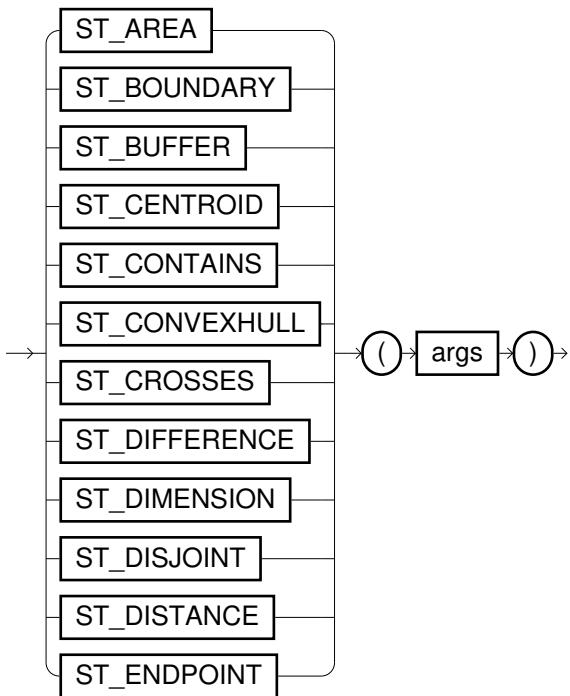
Several functions for geospatial objects. Please refer to [Section 2.4, “Geospatial data”](#) for details about geospatial objects and its functions.



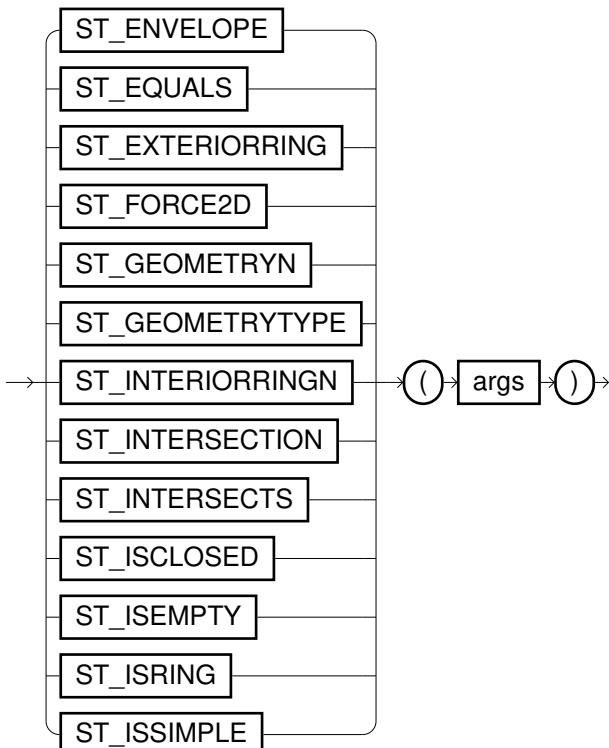
Please note that UDF scripts are part of the Advanced Edition of EXASOL.

Syntax

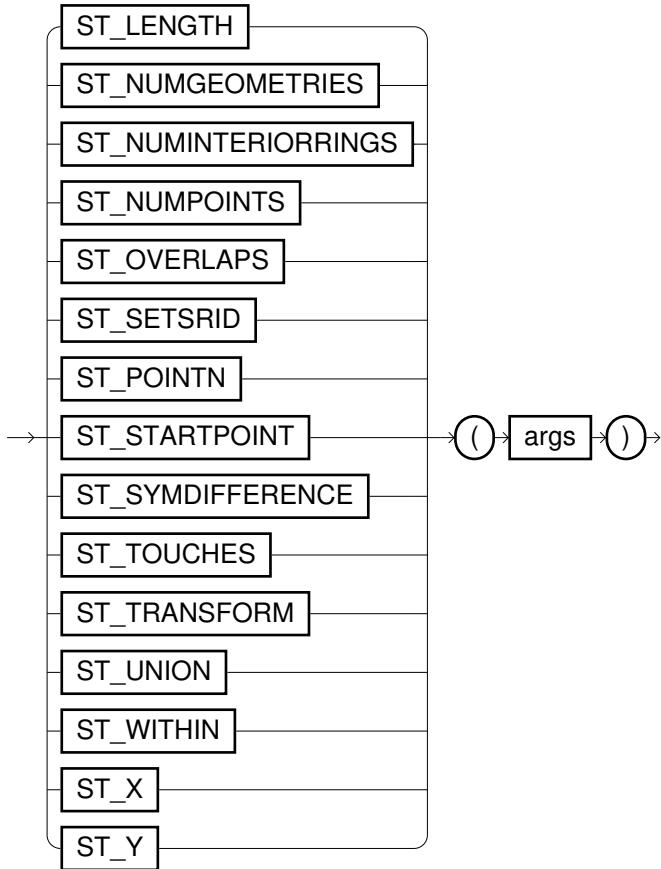
geospatial_functions_1 ::=



geospatial_functions_2 ::=



geospatial_functions_3 ::=



Example(s)

```

SELECT ST_DISTANCE('POLYGON ((0 0, 0 4, 2 4, 2 0, 0 0))',
                   'POINT(10 10)' ) ST_DISTANCE;

ST_DISTANCE
-----
10
  
```

STDDEV

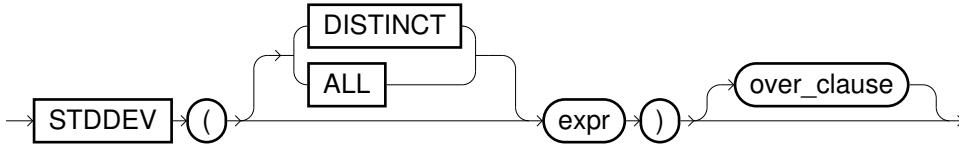
Purpose

Returns the standard deviation within a random sample. This equates to the following formula:

$$\text{STDDEV(expr)} = \sqrt{\frac{\sum_{i=1}^n (\text{expr}_i - \bar{\text{expr}})^2}{n - 1}}$$

Syntax

stddev::=



Note(s)

- Random samples with exactly one element have the result 0.
- If ALL or nothing is specified, then all of the entries are considered. If DISTINCT is specified, duplicate entries are only accounted for once.
- See also [Section 2.9.3, “Analytical functions”](#) for the OVER() clause and analytical functions in general.

Example(s)

```

SELECT STDDEV(salary) STDDEV FROM staff WHERE age between 20 and 30;

STDDEV
-----
19099.73821810132
  
```

STDDEV_POP

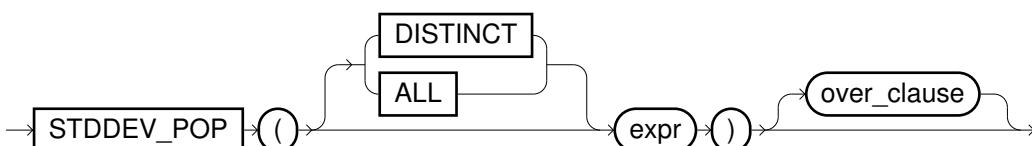
Purpose

Returns the standard deviation within a population. This equates to the following formula:

$$\text{STDDEV_POP(expr)} = \sqrt{\frac{\sum_{i=1}^n (\text{expr}_i - \bar{\text{expr}})^2}{n}}$$

Syntax

stddev_pop ::=



Note(s)

- If ALL or nothing is specified, then all of the entries are considered. If DISTINCT is specified, duplicate entries are only accounted for once.
- See also [Section 2.9.3, “Analytical functions”](#) for the OVER() clause and analytical functions in general.

Example(s)

```

SELECT STDDEV_POP(salary) STDDEV_POP FROM staff;
  
```

```

STDDEV_POP
-----
```

```
20792.12591343175
```

STDDEV_SAMP

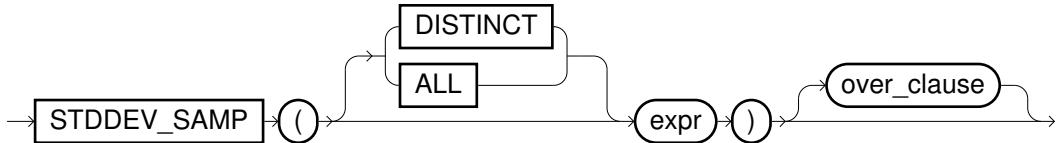
Purpose

Returns the standard deviation within a random sample. This equates to the following formula:

$$\text{STDDEV_SAMP(expr)} = \sqrt{\frac{\sum_{i=1}^n (\text{expr}_i - \bar{\text{expr}})^2}{n - 1}}$$

Syntax

`stddev_samp ::=`



Note(s)

- `STDDEV_SAMP` is identical to the `STDDEV` function. However, if the random sample only encompasses one element, the result is `NULL` instead of 0.
- If `ALL` or nothing is specified, then all of the entries are considered. If `DISTINCT` is specified, duplicate entries are only accounted for once.
- See also [Section 2.9.3, “Analytical functions”](#) for the `OVER()` clause and analytical functions in general.

Example(s)

```

SELECT STDDEV_SAMP(salary) AS STDDEV_SAMP
FROM staff WHERE age between 20 and 30;

STDDEV_SAMP
-----
19099.73821810132

```

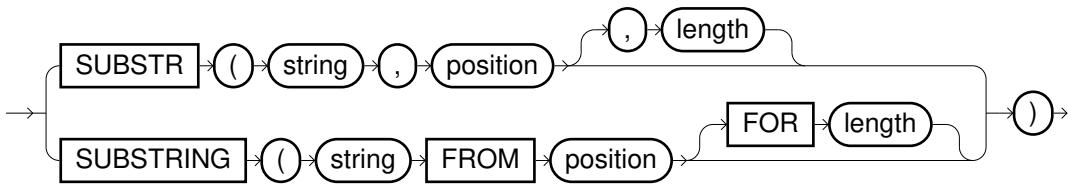
SUBSTR[ING]

Purpose

Returns a substring of the length `length` from the position `position`, out of the string `string`.

Syntax

`substring ::=`

**Note(s)**

- If `length` is not specified, all of the characters to the end of the string are used.
- If `position` is negative, counting begins at the end of the string.
- If `position` is 0 or 1, the result begins from the first character of the string.
- `MID` is an alias for this function.
- See also [REGEXP_SUBSTR](#).

Example(s)

```

SELECT SUBSTR('abcdef', 2, 3) S1,
       SUBSTRING('abcdef' FROM 4 FOR 2) S2
;

S1   S2
--- ---
bcd  de

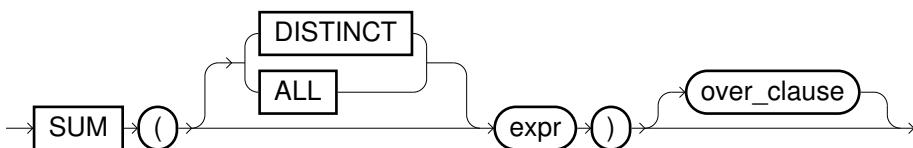
```

SUM**Purpose**

Returns the total.

Syntax

`sum ::=`

**Note(s)**

- If `ALL` or nothing is specified, then all of the entries are considered. If `DISTINCT` is specified, duplicate entries are only accounted for once.
- Only numeric operands are supported.
- See also [Section 2.9.3, “Analytical functions”](#) for the `OVER()` clause and analytical functions in general.

Example(s)

```

SELECT SUM(salary) SUM FROM staff WHERE age between 20 and 30;

```

```

SUM
-----
220500

SELECT SUM(salary) over (order by salary) SUM, salary FROM staff;

SUM          SALARY
-----
25000        25000
60500        35500
97500        37000
145500       48000
220500       75000
301500       81000

```

SYS_CONNECT_BY_PATH

Purpose

Returns for a CONNECT BY query a string containing the full path from the root node to the current node, containing the values for expr and separated by char. Details can be found in the description of the [SELECT statement](#) in [Section 2.2.4, “Query language \(DQL\)”](#).

Syntax

sys_connect_by_path::=

$$\rightarrow \boxed{\text{SYS_CONNECT_BY_PATH}} \rightarrow (\rightarrow \text{expr} \rightarrow , \rightarrow \text{char} \rightarrow) \rightarrow$$

Example(s)

```

SELECT SYS_CONNECT_BY_PATH(last_name, '/') "PATH"
  FROM employees
 WHERE last_name = 'Johnson'
 CONNECT BY PRIOR employee_id = manager_id
 START WITH last_name = 'Clark';

PATH
-----
/Clark/Jackson/Johnson

```

SYS_GUID

Purpose

Returns a system wide unique hexadecimal of type CHAR(48).

Syntax

sys_guid::=

→ **SYS_GUID** → () → ()

Example(s)

```
SELECT SYS_GUID();  
  
SYS_GUID()  
-----  
069a588869dfcaf8baf520c801133ab139c04f964f047f1
```

SYSDATE

Purpose

Returns the current system date by evaluating TO_DATE(SYSTIMESTAMP), thus interpreted in the current database time zone.

Syntax

sysdate ::=

→ **SYSDATE** →

Note(s)

- See also function [CURRENT_DATE](#).

Example(s)

```
SELECT SYSDATE;  
  
SYSDATE  
-----  
2000-12-31
```

SYSTIMESTAMP

Purpose

Returns the current timestamp, interpreted in the current database time zone.

Syntax

systimestamp ::=

→ **SYSTIMESTAMP** →

Note(s)

- The return value is of data type TIMESTAMP.
- More information about the database time zone can be found at function [DBTIMEZONE](#).
- Other functions for the current moment:
 - [CURRENT_TIMESTAMP](#) or [NOW](#)
 - [LOCALTIMESTAMP](#)

Example(s)

```
SELECT SYSTIMESTAMP;

SYSTIMESTAMP
-----
2000-12-31 23:59:59
```

TAN

Purpose

Returns the tangent of number n.

Syntax

tan ::=

→ **TAN** → (→ **n** →) →

Example(s)

```
SELECT TAN(PI() / 4);

TAN(PI() / 4)
-----
1
```

TANH

Purpose

Returns the hyperbolic tangent of number n.

Syntax

tanh ::=

→ **TANH** → (→ **n** →) →

Example(s)

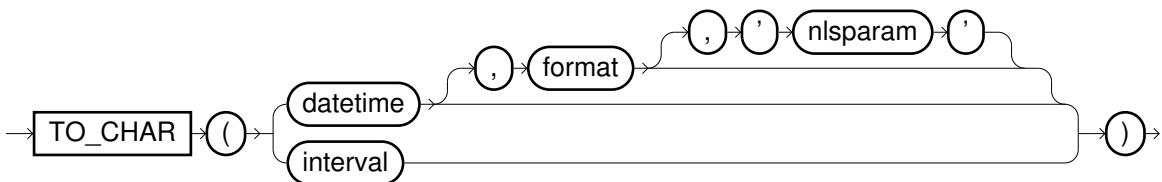
```
SELECT TANH(0) TANH;
TANH
-----
0
```

TO_CHAR (datetime)**Purpose**

Converts a date, timestamp or interval into a string.

Syntax

to_char (datetime)::=

**Note(s)**

- The standard format is used if no format is specified, this is defined in session parameter [NLS_DATE_FORMAT](#) or [NLS_TIMESTAMP_FORMAT](#).
- Possible formats can be found in [Section 2.6.1, “Date/Time format models”](#).
- Via the optional third parameter you can specify the language setting for the format (e.g. 'NLS_DATE_LANGUAGE=GERMAN'). Supported languages are German (DEU, DEUTSCH or GERMAN) and English (ENG, ENGLISH).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT TO_CHAR(DATE '1999-12-31') TO_CHAR;
TO_CHAR
-----
1999-12-31

SELECT TO_CHAR(TIMESTAMP '1999-12-31 23:59:00',
               'HH24:MI:SS DD-MM-YYYY') TO_CHAR;
TO_CHAR
-----
23:59:00 31-12-1999

SELECT TO_CHAR(DATE '2013-12-16',
               'DD. MON YYYY',
               'NLS_DATE_LANGUAGE=DEU') TO_CHAR;
```

TO_CHAR

16. DEZ 2013

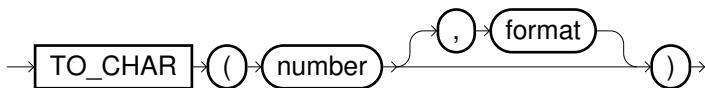
TO_CHAR (number)

Purpose

Converts a number into a string.

Syntax

to_char (number)::=



Note(s)

- Possible formats can be found in [Section 2.6.2, “Numeric format models”](#).

Example(s)

```
SELECT TO_CHAR(12345.6789) TO_CHAR;  
  
TO_CHAR  
-----  
12345.6789  
  
SELECT TO_CHAR(12345.67890, '9999999.99999999') TO_CHAR;  
  
TO_CHAR  
-----  
12345.678900000  
  
SELECT TO_CHAR(-12345.67890, '000G000G000D000000MI') TO_CHAR;  
  
TO_CHAR  
-----  
000,012,345.678900-
```

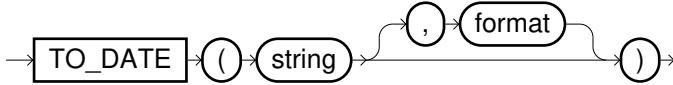
TO_DATE

Purpose

Converts a string into a date.

Syntax

to_date::=



Note(s)

- If no format is specified, the standard format is used to interpret `string`, this is defined in session parameter `NLS_DATE_FORMAT`.
- Possible formats can be found in [Section 2.6.1, “Date/Time format models”](#).
- ISO formats (IYYY, IW, ID) may not be mixed with non-ISO formats.
- If single elements are omitted, then their minimum values are assumed (e.g. `TO_DATE('1999-12', 'YYYY-MM')` is interpreted as December 1st, 1999).
- Session parameter `NLS_DATE_FORMAT` defines the output of the date.

Example(s)

```

SELECT TO_DATE('1999-12-31') TO_DATE;
TO_DATE
-----
1999-12-31

SELECT TO_DATE('31-12-1999', 'DD-MM-YYYY') TO_DATE;
TO_DATE
-----
1999-12-31
  
```

TO_DSINTERVAL

Purpose

Converts a string value into an interval (INTERVAL DAY TO SECOND).

Syntax

`to_dsinterval ::=`



Note(s)

- The string has always format `[+ -]DD HH:MI:SS[.FF]`. Valid values are 0-999999999 for days (DD), 0-23 for hours (HH), 0-59 for minutes (MI) and 0-59.999 for seconds (SS[.FF]).
- Please note that the fractional seconds are cut at the third position although you can specify more than three.
- See also [TO_YMINTERVAL](#), [NUMTODSINTERVAL](#) and [NUMTOYMINTEGRAL](#).

Example(s)

```

SELECT TO_DSINTERVAL('3 10:59:59.123') TO_DSINTERVAL;
  
```

```
TO_DSINTERVAL
-----
+000000003 10:59:59.123000000
```

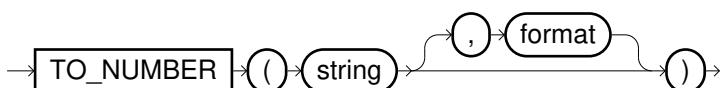
TO_NUMBER

Purpose

Converts a string into a number.

Syntax

to_number ::=



Note(s)

- The format has no influence on the value, but simply its representation.
- If a format is specified, the corresponding string may only contain digits as well as the characters plus, minus, `NLS_NUMERIC_CHARACTERS`, decimal point (decimal separator) and comma (group separator). However, plus and minus may only be used at the beginning of the string.
- The format for every digit in the string must contain format element nine or zero at least once, see also [Section 2.6.2, “Numeric format models”](#). If the string contains a decimal separator, the format must also contain the corresponding decimal separator element (D or .).
- If the data type of parameter `string` is no string, then the value is implicitly converted.
- The data type of the result of this function is dependent on the format, but typically a DECIMAL type. If no format is specified, the result type is DECIMAL(1, 0) in case of a boolean input parameter, and DOUBLE in any other case.

Example(s)

```
SELECT TO_NUMBER('+123') TO_NUMBER1,
       TO_NUMBER('-123.45', '99999.999') TO_NUMBER2;

TO_NUMBER1          TO_NUMBER2
-----
123      -123.450
```

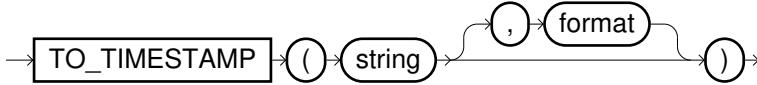
TO_TIMESTAMP

Purpose

Converts the string into a timestamp.

Syntax

to_timestamp ::=

**Note(s)**

- The standard format is used if no format is specified, this is defined in session parameter [NLS_TIMESTAMP_FORMAT](#).
- Possible formats can be found in [Section 2.6.1, “Date/Time format models”](#).
- Session parameter [NLS_TIMESTAMP_FORMAT](#) defines the output of the timestamp.

Example(s)

```

SELECT TO_TIMESTAMP('1999-12-31 23:59:00') TO_TIMESTAMP;
TO_TIMESTAMP
-----
1999-12-31 23:59:00.000000

SELECT TO_TIMESTAMP('23:59:00 31-12-1999',
                   'HH24:MI:SS DD-MM-YYYY') TO_TIMESTAMP;

TO_TIMESTAMP
-----
1999-12-31 23:59:00.000000
  
```

TO_YMINTERVAL**Purpose**

Converts a string value into an interval (INTERVAL YEAR TO MONTH).

Syntax

to_yminterval ::=

**Note(s)**

- The string always has format [+ | -]YY-MM. Valid values are 0 to 999999999 for years (YY) and 0 to 11 for months (MM).
- See also [TO_DSINTERVAL](#), [NUMTODSINTERVAL](#) and [NUMTOYMINTEGRAL](#).

Example(s)

```

SELECT TO_YMINTERVAL('3-11') TO_YMINTERVAL;
TO_YMINTERVAL
-----
+000000003-11
  
```

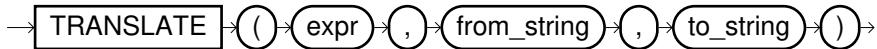
TRANSLATE

Purpose

Replaces the characters out of `from_string` with the corresponding character out of `to_string` in the string, `expr`.

Syntax

`translate ::=`



Note(s)

- Those characters in `expr` which do not exist in `from_string` are not replaced.
- If `from_string` is longer than `to_string`, the relevant characters are deleted and not replaced.
- If `to_string` is longer than `from_string`, the relevant characters are ignored during the replacement process.
- If one of the parameters is the empty string, then the result is NULL.

Example(s)

```
SELECT TRANSLATE( 'abcd' , 'abc' , 'xy' ) TRANSLATE;
TRANSLATE
-----
xyd
```

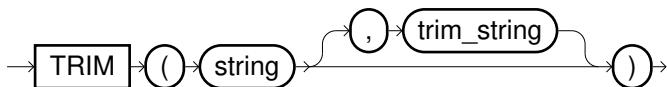
TRIM

Purpose

TRIM deletes all of the characters specified in the expression, `trim_string`, from both the right and left border of `string`.

Syntax

`trim ::=`



Note(s)

- If `trim_string` is not specified, spacing characters are deleted.

Example(s)

```
SELECT TRIM('abcdef', 'acf') "TRIM";
-----  
bcde
```

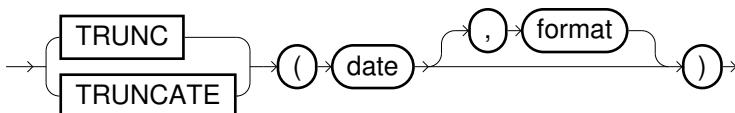
TRUNC[ATE] (datetime)

Purpose

Returns a date and/or a timestamp, which is trimmed in accordance with the format definition. Accordingly, TRUNC(datetime) behaves in exactly the same way as [ROUND\(datetime\)](#), with the exception that TRUNC rounds-down.

Syntax

trunc(datetime)::=



Note(s)

- The following elements can be used as format:

CC, SCC	Century
YYYY, SYYY, YEAR, SYEAR,	Year
YY, YY, Y	
IYYY, IYY, IY, I	Year in accordance with international standard, ISO 8601
Q	Quarter
MONTH, MON, MM, RM	Month
WW	Same day of the week as the first day of the year
IW	Same day of the week as the first day of the ISO year
W	Same day of the week as the first day of the month
DDD, DD, J	Day
DAY, DY, D	Starting day of the week. The first day of a week is defined via the parameter NLS_FIRST_DAY_OF_WEEK (see ALTER SESSION and ALTER SYSTEM).
HH, HH24, HH12	Hour
MI	Minute
SS	Seconds
- A similar functionality provides the PostgreSQL compatible function [DATE_TRUNC](#).
- If a format is not specified, the value is truncated to days.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT TRUNC(DATE '2006-12-31', 'MM') TRUNC;
-----  
TRUNC
```

```
-----
2006-12-01

SELECT TRUNC(TIMESTAMP '2006-12-31 23:59:59', 'MI') TRUNC;

TRUNC
-----
2006-12-31 23:59:00.000000
```

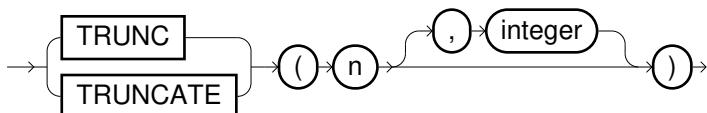
TRUNC[ATE] (number)

Purpose

Trims number n to integer places behind the decimal point.

Syntax

trunc (number)::=



Note(s)

- If the second argument is not specified, trimming is conducted to a whole number.
- If the second argument is negative, trimming is conducted to integer digits in front of the decimal point.

Example(s)

```
SELECT TRUNC(123.456,2) TRUNC;

TRUNC
-----
123.45
```

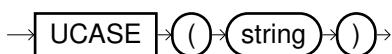
UCASE

Purpose

Converts the specified string into upper case letters.

Syntax

ucase::=



Note(s)

- UCASE is an alias for [UPPER](#).

Example(s)

```
SELECT UCASE( 'AbCdEf' ) UCASE;  
  
UCASE  
-----  
ABCDEF
```

UNICODE

Purpose

Returns the numeric unicode value of a character.

Syntax

unicode ::=

→ **UNICODE** → (→ **char** →) →

Note(s)

- See also [UNICODECHR](#).

Example(s)

```
SELECT UNICODE( 'ä' ) UNICODE;  
  
UNICODE  
-----  
228
```

UNICODECHR

Purpose

Returns the unicode character which equates to the numeric value n .

Syntax

unicodechr ::=

→ **UNICODECHR** → (→ **n** →) →

Note(s)

- The number n has to be between 0 and 2097151.
- `UNICODECHR(0)` returns `NULL`.
- See also [UNICODE](#).

Example(s)

```
SELECT UNICODECHR( 252 )  UNICODECHR;  
  
UNICODECHR  
-----  
ü
```

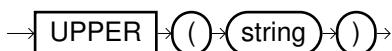
UPPER

Purpose

Converts the specified string into upper case letters.

Syntax

upper ::=



Note(s)

- `UPPER` is an alias for [UCASE](#).

Example(s)

```
SELECT UPPER( 'AbCdEf' )  UPPER;  
  
UPPER  
-----  
ABCDEF
```

USER

Purpose

Returns the current user.

Syntax

user ::=



Note(s)

- This is equivalent to [CURRENT_USER](#).

Example(s)

```
SELECT USER;
USER
-----
SYS
```

VALUE2PROC**Purpose**

This function returns the corresponding database node for a certain value. This mapping corresponds with the data distribution that would be achieved if you do a `DISTRIBUTE BY` that value.

Syntax

`value2proc ::=`

→ **VALUE2PROC** → (→ **expr** →) →

Note(s)

- This function can be used to understand the actual data distribution across the cluster nodes which can be achieved with the [ALTER TABLE \(distribution\)](#).
- The return value is an integer between 0 and [NPROC-1](#).
- In this context, please also note functions [IPROC](#) and [NPROC](#).

Example(s)

```
SELECT IPROC(),
       c1, VALUE2PROC(c1) V2P_1,
       c2, VALUE2PROC(c2) V2P_2 FROM t;

IPROC C1 V2P_1 C2          V2P_2
----- -----
0      1   3     abc      3
1      2   2     abcd     0
2      3   1     abcde    1
3      4   0     abcdef   3
0      5   3     abcdefg  0
1      6   2     abcdefgh 2
2      7   2     abcdefghi 1
3      8   1     abcdefghij 1
```

VAR_POP

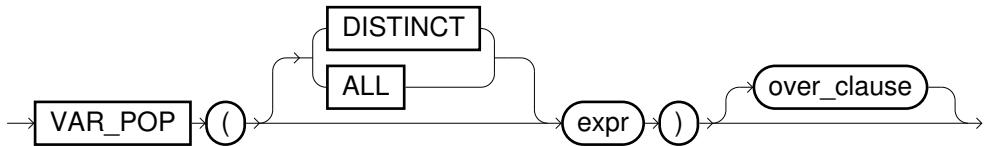
Purpose

Returns the variance within a population. This equates to the following formula:

$$\text{VAR_POP(expr)} = \frac{\sum_{i=1}^n (\text{expr}_i - \bar{\text{expr}})^2}{n}$$

Syntax

`var_pop ::=`



Note(s)

- If ALL or nothing is specified, then all of the entries are considered. If DISTINCT is specified, duplicate entries are only accounted for once.
- See also [Section 2.9.3, “Analytical functions”](#) for the OVER() clause and analytical functions in general.

Example(s)

```

SELECT VAR_POP(salary) AS VAR_POP FROM staff;
VAR_POP
-----
432312500
  
```

VAR_SAMP

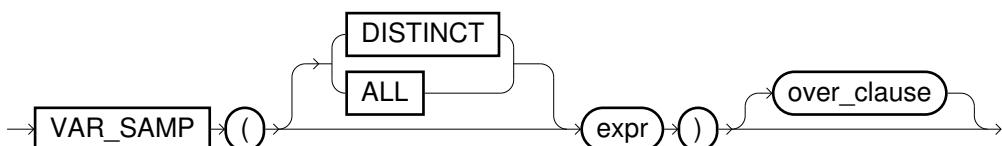
Purpose

Returns the variance within a random sample. This equates to the following formula:

$$\text{VAR_SAMP(expr)} = \frac{\sum_{i=1}^n (\text{expr}_i - \bar{\text{expr}})^2}{n - 1}$$

Syntax

`var_samp ::=`



Note(s)

- `VAR_SAMP` is identical to the `VARIANCE` function. However, if the random sample only encompasses one element, the result is `NULL` instead of 0.
- If `ALL` or nothing is specified, then all of the entries are considered. If `DISTINCT` is specified, duplicate entries are only accounted for once.
- See also [Section 2.9.3, “Analytical functions”](#) for the `OVER()` clause and analytical functions in general.

Example(s)

```
SELECT VAR_SAMP(salary) AS VAR_SAMP FROM staff WHERE age between 20 and 30;
VAR_SAMP
-----
364800000
```

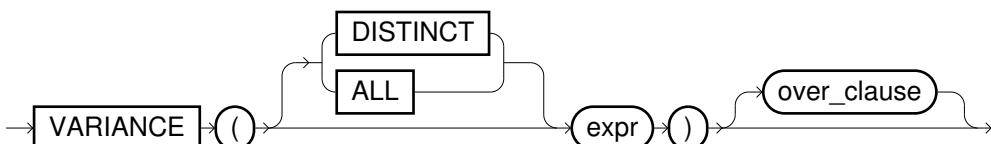
VARIANCE**Purpose**

Returns the variance within a random sample. This equates to the following formula:

$$\text{VARIANCE(expr)} = \frac{\sum_{i=1}^n (\text{expr}_i - \bar{\text{expr}})^2}{n - 1}$$

Syntax

`variance ::=`

**Note(s)**

- Random samples with exactly one element have the result 0.
- If `ALL` or nothing is specified, then all of the entries are considered. If `DISTINCT` is specified, duplicate entries are only accounted for once.
- See also [Section 2.9.3, “Analytical functions”](#) for the `OVER()` clause and analytical functions in general.

Example(s)

```
SELECT VARIANCE(salary) VARIANCE FROM staff WHERE age between 20 and 30;
VARIANCE
-----
364800000
```

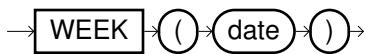
WEEK

Purpose

Returns the week of a date (values 1-53, specified in ISO-8601 standard).

Syntax

week ::=



Note(s)

- A new week generally begins with Monday.
- The first week of a year starts from January 1st if it is a Monday, Tuesday, Wednesday or Thursday - otherwise on the following Monday.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT WEEK(DATE '2012-01-05') WEEK;  
  
WEEK  
----  
1
```

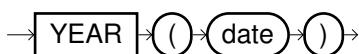
YEAR

Purpose

Returns the year of a date.

Syntax

year ::=



Note(s)

- This function can also be applied on strings, in contrast to function [EXTRACT](#).
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT YEAR(DATE '2010-10-20');
```

```
YEAR(
-----
2010
```

YEARS_BETWEEN

Purpose

Returns the number of years between two date values.

Syntax

years_between ::=

→ **YEARS_BETWEEN** → (→ **datetime1** → , → **datetime2** →) →

Note(s)

- If a timestamp is entered, only the date contained therein is applied for the computation.
- If the months and days are identical, the result is an integer.
- If the first date value is earlier than the second date value, the result is negative.
- For data type TIMESTAMP WITH LOCAL TIME ZONE this function is calculated within the session time zone.

Example(s)

```
SELECT YEARS_BETWEEN(DATE '2001-01-01', DATE '2000-06-15') YB1,
       YEARS_BETWEEN(TIMESTAMP '2001-01-01 12:00:00',
                      TIMESTAMP '2000-01-01 00:00:00') YB2;

YB1          YB2
-----
0.5456989247312  1
```

ZEROIFNULL

Purpose

Returns 0 if number has value NULL. Otherwise, number is returned.

Syntax

zeroifnull ::=

→ **ZEROIFNULL** → (→ **number** →) →

Note(s)

- The ZEROIFNULL function is equivalent to the CASE expression CASE WHEN number is NULL THEN 0 ELSE number END.

- See also function [NULLIFZERO](#).

Example(s)

```
SELECT ZEROIFNULL(NULL) ZIN1, ZEROIFNULL(1) ZIN2;
```

ZIN1	ZIN2
0	1

Chapter 3. Concepts

This chapter introduces some fundamental concepts in Exasol.

3.1. Transaction management

Exasol ensures multi-user capability through the implementation of a transaction management system (TMS). This means that various requests from different users can be processed in parallel. This chapter contains an explanation of the basic concept of transactions as well as tips and recommendations for working with the Exasol TMS.

3.1.1. Basic concept

A transaction consists of several SQL statements. These are either confirmed with a **COMMIT** statement or undone with a **ROLLBACK** statement.

Example

```
-- Transaction 1
CREATE SCHEMA my_schema;
COMMIT;

-- Transaction 2
CREATE TABLE t (i DECIMAL);
SELECT * FROM t;
ROLLBACK;

-- Transaction 3
CREATE TABLE t (i VARCHAR(20));
COMMIT;
```

The aim of a transaction-based system is the maintenance of complete transaction security, i.e. each transaction should return a correct result and leave the database in a consistent condition. To achieve this, transactions must comply with the so-called ACID principles:

- Atomicity – Transactions are either fully executed, or not at all.
- Consistency – The transaction is given the internal consistency of the database.
- Isolation – The transaction is executed as if it is the only transaction in the system.
- Durability – All changes to a completed transaction confirmed with COMMIT remain intact.

In order to ensure compliance with the ACID principles, every transaction is subject to an evaluation by the TMS. If necessary, the TMS intervenes and automatically rectifies conflicts through the enforcement of waiting times or by rolling back transactions in the event of a collision.



Information about occurring transaction conflicts can be found in system tables **EXA_USER_TRANSACTION_CONFLICTS_LAST_DAY** and **EXA_DBA_TRANSACTION_CONFLICTS**.

In order to keep the number of colliding transactions as low as possible, Exasol supports the so-called "MultiCopy Format". This means that multiple versions of every database object may exist (temporarily). In this manner system throughput (number of fully executed transactions per unit of time) can be significantly increased compared to databases with SingleCopy format.

The individual transactions are isolated from one another by the TMS by means of a lock procedure. The granularity of the lock procedure always surrounds an entire database object, e.g. one schema or one table. This means, for example, that two transactions cannot simultaneously update different rows of a table.

Due to the TMS, for each transaction that is started the user is confronted with one of the following scenarios:

1. The transaction is run to the end.
2. The transaction is run to the end, but waiting times occur due to a requirement to wait for other transactions to finish.
3. The transaction cannot be completed due to collisions with other transactions and is rolled back. In this case, the user can repeat the transaction later.

3.1.2. Differences to other systems

Certain other database systems have only partially implemented the transaction model and sometimes conceal transactions from the user. For instance, the system may directly store schema statements (e.g. "CREATE SCHEMA", "CREATE TABLE") persistently in the database.

This reduces the risk of a collision when simultaneously executing transactions with schema statements, but has the disadvantage that a rollback, for example, cannot undo the schema statement that has been executed.

In contrast, the Exasol TMS does not conceal transactions from the user and does not persistently store statements in the database automatically.

3.1.3. Recommendations for the user

In order to minimize the risk of transaction conflicts during parallel access by very many users, the interfaces to Exasol (EXAplus and drivers) have the option "Autocommit = ON" set by default. In AUTOCOMMIT mode, successfully completed SQL statements are automatically saved persistently.

However, for parts that affect the performance of the system, it may be more effective to not run a COMMIT after each SQL statement. This is especially true if intermediate tables, which are not intended to be saved persistently, are computed in scripts. Therefore, in EXAplus the user has the possibility of disabling this option with the command, "SET AUTOCOMMIT OFF".



AUTOCOMMIT = OFF increases the risk of collisions and can affect other users negatively.

If the autocommit mode is disabled, the option "-x" in the EXAplus console is recommended. This causes a SQL script to be aborted if an error occurs, particularly during an automatic rollback after a transaction conflict. If batch scripts are started without this option, processing of the SQL statement sequence would continue despite the error, which could lead to incorrect results and should, therefore, be avoided.

3.2. Rights management

The security of and access to the database can be controlled using the **Data Control Language (DCL)**. Through the management of users and roles as well as the granting of privileges, it can be determined (fine-grained) who is permitted to perform what actions in the database.

The following SQL statements are components of the DCL:

CREATE USER	Creates a user
ALTER USER	Changes the password
DROP USER	Deletes a user
CREATE ROLE	Creates a role
DROP ROLE	Deletes a role
GRANT	Grants roles, system privileges and object privileges to users or roles
REVOKE	Withdraws roles, system privileges and object privileges from users or roles
ALTER SCHEMA	Changes the owner of a schema (and thus all its schema objects)

This chapter is merely an introduction to the basic concepts of rights management. Full details are set out in [Appendix B, Details on rights management](#) and [Section 2.2.3, “Access control using SQL \(DCL\)”](#).

3.2.1. User

An administrator must create a **USER** account for each user who wishes to connect to the database (with the **CREATE USER** SQL statement). In the process, the user receives a password that can be changed at any time and with which he authenticates himself to the database.

The naming conventions for user names and passwords are the same as for SQL identifiers (identifier for database objects such as table names, see also [Section 2.1.2, “SQL identifier”](#)). However, with this case sensitivity is of no significance.



User names and roles are not case sensitive

Appropriate privileges are necessary for a user to perform actions in the database. These are granted or withdrawn by an administrator or other users. For example, a user needs the system privilege **CREATE SESSION** in order to connect to the database. This system privilege is withdrawn if one wishes to temporarily disable a user.

A special user exists in the database, **SYS**, this cannot be deleted and it possesses universal privileges.



Initially the SYS password is *exasol*, however, this should be changed immediately after the first login in order to prevent potential security risks.

3.2.2. Roles

Role facilitate the grouping of users and simplify rights management considerably. They are created with the **CREATE ROLE** statement.

A user can be assigned one or several roles with the **GRANT** SQL statement. This provides the user with the rights of the respective role. Instead of granting many "similar" users the same privileges, one can simply create a role and grant this role the appropriate privileges. By assigning roles to roles, a hierarchical structure of privileges is even possible.

Roles cannot be disabled (as with, e.g. Oracle). If an assignment to a role needs to be reversed, the role can be withdrawn by using the **REVOKE** SQL statement.

The **PUBLIC** role stands apart because every user receives this role automatically. This makes it very simple to grant and later withdraw certain privileges to/from all users of the database. However, this should only occur if

one is quite sure that it is safe to grant the respective rights and the shared data should be publicly accessible. The PUBLIC role cannot be deleted.

Another pre-defined role is the **DBA** role. It stands for database administrator and has all possible privileges. This role should only be assigned to very few users because it provides these with full access to the database. Similar to PUBLIC, the DBA role cannot be deleted.

3.2.3. Privileges

Privileges control access on the database. Privileges are granted and withdrawn with the SQL statements, **GRANT** and **REVOKE**. Distinction is made between two different privilege types:

System privileges control general rights such as "Create new schema", "Create new user" or "Access any table".

Object privileges allow access to single schema objects (e.g. "SELECT access to table t in schema s"). Tables, views, functions and scripts are referred to as schema objects. It should be noted that each schema and all the schema objects contained therein belong to exactly one user or one role. This user and all owners of this role have the right to delete these objects and grant other users access to them. If an object privilege is granted for a schema, then this privilege is applied to all containing schema objects.

A detailed list of all privileges available in Exasol can be found in [Section B.1, “List of system and object privileges”](#).

In order to be able to grant or withdraw privileges to/from a user or a role, the user himself must possess certain privileges. The exact rules are explained in the detailed description of the GRANT and REVOKE statements in the SQL reference (see also [Section 2.2.3, “Access control using SQL \(DCL\)”](#)).

One should always be aware of whom one grants what privileges and what the potential consequences are of doing so. In this respect, particular emphasis is given to the system privileges: **GRANT ANY ROLE**, **GRANT ANY PRIVILEGE** and **ALTER USER**. They allow full access to the database and should only be granted to a limited number of users. Through the system privilege, GRANT ANY ROLE, a user can assign the DBA role to any other user (naturally, also himself) and in doing so would have full access to the database. If the user has the GRANT ANY PRIVILEGE system privilege, he can grant himself or any other user the GRANT ANY ROLE system privilege and in turn receive full access. With the ALTER USER privilege it is possible to change the SYS password and in doing so also receive full access.



The GRANT ANY ROLE, GRANT ANY PRIVILEGE, and ALTER USER system privileges provide full access to the database.

3.2.4. Access control with SQL statements

Before any SQL statement is executed a check of whether the current user has appropriate rights is made. If this is not the case, an error message is displayed.

An overview of all SQL statements supported by Exasol as well as the necessary privileges can be found in [Section B.2, “Required privileges for SQL statements”](#). In [Chapter 2, SQL reference](#) in the detailed description of each SQL statement, this information is also provided.

Access rights to the individual columns of a table or view is not supported. If only part of a table should be visible for certain users/roles, this can be achieved by using views, which select the relevant part. Instead of granting access to the actual table, this is only permitted for the generated view. This allows access protection for specific columns and/or rows.

3.2.5. Meta information on rights management

The status of rights management in the database can be queried from numerous **system tables**. Information on the existing users and roles as well as their rights is contained in these. Additionally, the user is able to clarify his roles, the schema objects to which he has access, the privileges he has granted to others and his own privileges.

A list of all the system tables relevant to rights management can be found in [Section B.3, “System tables for rights management”](#).

Who has access to what system tables is also controlled by privileges. There are some, to which only a DBA has access for reasons of security. In addition, there are system tables that are visible to all but that only show individually permitted information (for example, **EXA_ALL_OBJECTS**: all schema objects to which the user has access).

3.2.6. Rights management and transactions

Users, roles, and privileges are based on transactions in exactly the same way as the schema objects of the database.

This means that changes are not visible until the transaction has been confirmed by means of COMMIT. Due to the fact that with all SQL statements a read operation on the user's privileges is performed, where possible **DCL** statements should always be conducted with AUTOCOMMIT switched on. If not, there is an increase in the risk of transaction conflicts.

More information on this issue can be found in [Section 3.1, “Transaction management”](#).

3.2.7. Example of rights management

The following example scenario is designed to illustrate the basic mechanisms of rights management. The following requirements should be implemented:

- Role, ANALYST: performs analyzes on the database, therefore he is permitted to read all tables and create his own schemas and schema objects
- Role, HR: manages the staff, therefore he is permitted to edit the STAFF table
- Role, DATA_ADMIN: gives full access to the data schema
- Table, STAFF: contains information on the company's staff
- User, SCHNEIDER: is an administrator and can do anything
- User, SCHMIDT: works in marketing and has the ANALYST role
- User, MAIER: works in the personnel office, therefore he has the HR role
- User, MUELLER: is an administrator for the DATA schema and can only gain access to this. Therefore, he owns the DATA_ADMIN role

The following SQL script could be used to implement this scenario in the database:

```
--create table
CREATE SCHEMA infos;
CREATE TABLE infos.staff (id          DECIMAL,
                         last_name  VARCHAR(30),
                         name       VARCHAR(30),
                         salary     DECIMAL);

CREATE SCHEMA data_schema;

--create roles
CREATE ROLE analyst;
CREATE ROLE hr;
CREATE ROLE data_admin;

--create users
```

```
CREATE USER schneider IDENTIFIED BY s56_f;
CREATE USER schmidt IDENTIFIED BY x234aj;
CREATE USER maier IDENTIFIED BY jd89a2;
CREATE USER mueller IDENTIFIED BY lk9a4s;

--for connecting to db
GRANT CREATE SESSION TO schneider, schmidt, maier, mueller;

--grant system privileges to role analyst
GRANT CREATE SCHEMA, CREATE TABLE, SELECT ANY TABLE TO analyst;

--grant object privileges on one table to role hr
GRANT SELECT,UPDATE,DELETE ON TABLE infos.staff TO hr;

--grant system privileges to role data_admin
GRANT CREATE TABLE, CREATE VIEW TO data_admin;

--make data_admin the owner of schema data
ALTER SCHEMA data_schema CHANGE OWNER data_admin;

--grant roles to users
GRANT dba TO schneider;
GRANT analyst TO schmidt;
GRANT hr TO maier;
GRANT data_admin TO mueller;
```

3.3. Priorities

3.3.1. Introduction

By the use of priorities, Exasol resources can be systematically distributed across users and roles.

Even during the execution of one single query, Exasol attempts to use of as many resources (CPU, RAM, Network, I/O) as possible by internal parallelization (multithreading). But since not all execution steps can be parallelized, the utilization of all hardware resources will be achieved only with multiple parallel queries. If more parallel queries are executed than the number of cores per server, Exasol's resource manager schedules the parallel queries to ensure a constant system performance.

Using a time slot model, the resource manager distributes the resources evenly across the parallel queries while limiting the overall resource consumption by regularly pausing and restarting certain queries. Without such scheduling the overall usage could exhaust the system resources, leading to a decreased throughput. Without explicitly setting priorities, Exasol treats all queries equally except short queries running less than 5 seconds, which get a higher weighting for minimizing the system's latency.

If you want to influence the execution of many parallel queries, you can use the priorities which can be assigned via the [GRANT](#) statement. This intervention should only be necessary in case of a highly parallel usage of your system, combined with a certain user group which needs maximal performance. Priorities could be for instance useful if you connect a web application which has low latency requirements, and your system frequently has to process 10 or more parallel queries. On the other side, long running ETL processes or queries from less important applications could get a lower priority.



Valuable information about the usage of your database can be found in the statistical system tables (e.g. [EXA_USAGE_DAILY](#) and [EXA_SQL_DAILY](#), see also [Section A.2.3, “Statistical system tables”](#)).

3.3.2. Priorities in Exasol

In default case, the sessions of all users have the same priority. But if you identified certain users or roles which should be prioritized higher or lower, then you can chose between the priority groups **LOW**, **MEDIUM** and **HIGH** with weights 1, 3 and 9. These three groups define the resource distribution by allocating at least about 8%, 23% and 69% in accordance to their weights. Users within one group share these resources (CPU and RAM) equally. If a priority group is not represented at all, then the resources are distributed across the other groups by their weights. If e.g. only LOW and MEDIUM sessions exist, then these obtain 25% and 75% of the resources (corresponding to their weights 1:3).

Notes

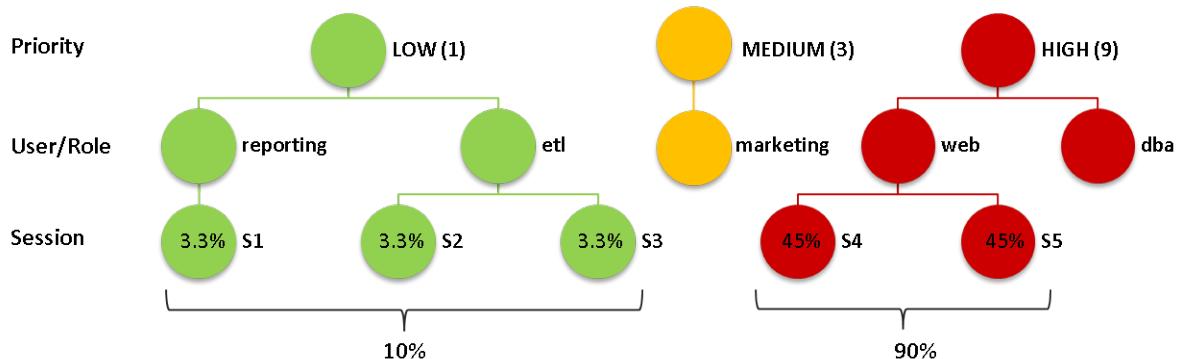
- Users without an explicit priority obtain the default priority group MEDIUM.
- A user inherits the highest priority of his roles, but a directly granted priority overwrites that.
- Multiple sessions of a certain user obtain the same priority.
- Please note that all sessions of a priority group share its resources equally. As consequence, e.g. each of many parallel HIGH sessions can get less resources than a single LOW session. On the other hand, this way of system allocation has the great advantage that certain user groups can be assured to get a certain amount of resources (e.g. at least 69% for the HIGH users).
- The execution times of queries doesn't exactly follow the resource distribution. Hence an identical query with twice as much resources than another one will be executed significantly faster, but not exactly by factor 1:2.
- A user can set the NICE attribute via the statement [ALTER SESSION](#) (`ALTER SESSION SET NICE='ON'`). This indicates the system that the sessions of other users shall be affected as less as possible by this user's session. The resource manager then divides the session's weight by the number of currently active queries. Hereby such sessions can be processed without affecting sessions from the same or other priority groups.

- The priority and concrete resource allocation of users, roles and their sessions can be displayed via several system tables (e.g. `EXA_ALL_USERS`, `EXA_ALL_ROLES`, `EXA_ALL_SESSIONS`).

3.3.3. Example

In the following example some roles with different priorities are defined. In the picture below, the resulting resource distribution for a list of active sessions is displayed.

```
GRANT PRIORITY LOW      TO reporting, etl;  
GRANT PRIORITY MEDIUM  TO marketing;  
GRANT PRIORITY HIGH    TO web, dba;
```



3.4. ETL Processes

3.4.1. Introduction

ETL processes are the transfer of data from source systems into a target system:

- **Extract:** the reading of data from the source systems
- **Transform:** various data modifications, customization of data types, elimination of duplicates, aggregation, standardization of heterogeneous data structures
- **Load:** writing data in the target system

With regards to the source systems, these usually involve one or more operational databases, the data of which are to be combined in a target system. This normally begins with an initial load; the periodic loading of data then occurs incrementally in order to keep the data warehouse up-to-date.

ETL or ELT?

Instead of using ETL tools, more complex data transformations than just data type conversions can of course also be performed directly within the database using SQL statements. This includes for example data modifications, aggregation or schema manipulations.

One can also speak of ELT (instead of ETL) processes in this context, because the Transformation phase is performed after the Extract and Load. This approach is recommended by Exasol because the high performance of the parallel cluster and the database can be exploited for complex operations.

3.4.2. SQL commands IMPORT and EXPORT

Loading processes in Exasol are controlled with the **IMPORT** and **EXPORT** statements. A detailed description of the syntax and examples can be found in [Section 2.2.2, “Manipulation of the database \(DML\)”](#).

Instead of an external bulk loading tool, Exasol's loading engine is directly integrated within the cluster. This provides optimal performance by leveraging a powerful, parallel cluster, and simplifies the overall ETL process by just using SQL statements instead of having to install and maintain client tools on different platforms and systems.

The credentials for external systems can easily be encapsulated by using connection objects (see also [CREATE CONNECTION](#) in [Section 2.2.3, “Access control using SQL \(DCL\)”](#)).

Integration of data management systems

You can integrate nearly any data management system directly into the IMPORT and EXPORT statements by just uploading the corresponding JDBC driver into the Exasol cluster (via EXAoperation) and use the generic JDBC interface for IMPORT and EXPORT. And by supporting native interfaces to the database systems Exasol and Oracle, we achieve even higher performance.

Therefore, the data integration from other databases or data management systems is very simple, powerful and flexible. You can load data by a simple SQL command and combine that with further post-processing via subselects. You can even specify a statement instead of a table name for the external data source which is executed on that system, e.g. to just load a certain part of a table into Exasol.

File load/export

For pure file processing, both the **CSV** (Comma separated Value) and the **FBV** (Fix Block Value) file formats are supported which are described later on. You achieve the best performance if you specify an HTTP or FTP server and even split the files into several parts to allow Exasol to read/write the data in parallel.

In case you want to read from or write into local files of your computer, you can also use the statements **IMPORT** and **EXPORT**, but only in combination with EXAplus or the JDBC driver.

In some cases, the integration of an external bulk loading tool is still necessary, and for these cases we deliver the tool `exajload` as part of our JDBC driver package.

3.4.3. Scripting complex ETL jobs

IMPORT and **EXPORT** statements are usually part of a more complex logic to update the data. Sometimes, old data is deleted or new and changed data is merged afterwards, or even more complex processes such as data cleansing is performed. And to create flexible ETL processes, it is often necessary to react to exceptions or run different execution branches depending on certain conditions.

That's why we recommend to read [Section 3.5, “Scripting”](#) if you plan to develop a more sophisticated ETL process. With our scripting capabilities, you can execute several SQL statements within one script, use exception handling and even process smaller amounts of data directly within the programming language.

3.4.4. User-defined IMPORT using UDFs

In case the standard interfaces, such as files, JDBC sources or natively Oracle and Exasol, are not sufficient for your ETL scope, you can develop user-defined ETL processes with the help of UDF scripts (see also [Section 3.6, “UDF scripts”](#)). In the following we are going to explain how extremely flexible this capability is and how you can integrate nearly any system and data format into Exasol (e.g. Hadoop).

The foundation of this concept is the creation of a row-emitting UDF script (i.e. with return type **EMIT**) by using the **CREATE SCRIPT** statement. If you integrate such a script within an `INSERT INTO <table> (SELECT my_emit_script('param') [FROM . . .])` command, then this UDF script is called on the cluster during the execution of the SQL statement and the emitted rows will be inserted into the target table.

Within the script code you can implement all possible processes, such as establishing connections to external systems, extracting the data, data type conversion and even more complex transformations. Via parameters you can hand over all necessary information to the script.

The big advantage of this architecture is the capability to integrate all kinds of (open source) libraries from various script languages to perform specific functions. For example: if you use libraries from a script language to take care of data formats, you can easily use new data formats from Hadoop systems within Exasol, without needing to wait for the next software release.

By leveraging the ability to use dynamic input and output parameters (see also [Dynamic input and output parameters](#)) you can develop scripts generically which can be re-used for all kinds of tables with different structures and data types.

Using UDFs to load data in parallel

To achieve optimal loading performance you have the option to parallelize ETL processes using UDF scripts by specifying the **GROUP BY** clause. By that, the UDF script is called in parallel for every single group, exactly as it is in the normal usage of UDF scripts within plain **SELECT** statements.

The following example demonstrates how easy it is to load data in parallel with the help of UDF scripts. The inner **SELECT** invokes a UDF script (once) which connects to a service and returns the list of existing JSON files, plus a certain partition number. The outer **SELECT** calls another UDF script (with input type **SET**) that finally loads that data in parallel, according to partition ID.

```
INSERT INTO my_table (
  SELECT load_my_json(service_address, filepath) EMITS (revenues DECIMAL)
```

```
FROM (SELECT get_json_files('my-service-address', 10))
GROUP BY partition);
```

IMPORT FROM SCRIPT syntax

Using UDF scripts within INSERT commands can get very complex and confusing. That's why we added the IMPORT FROM SCRIPT syntax for better usability (details about the syntax can be found here: [IMPORT](#)).

The only precondition is that the UDF script implements a specific method that creates a corresponding SELECT statement string out of the IMPORT information. In the simplest case, the same UDF script will be called within that SELECT that invokes the actual data loading. Specifics about this method for the various script languages can be found in [Section 3.6.3, “Details for different languages”](#).

Therefore the internal execution consists of two phases, the SQL text generation and the subsequent embedding into an INSERT INTO SELECT statement. Exasol handles the second phase by considering and adds the target table name and columns.

The following simple example illustrates the general concept. The user-defined IMPORT generates a sequence of integer values whose size is defined via a parameter.

```
CREATE PYTHON SCALAR SCRIPT etl.generate_seq (num INT) EMITS (val INT) AS
def generate_sql_for_import_spec(import_spec):
    if not "RECORDS" in import_spec.parameters:
        raise ValueError('Mandatory parameter RECORDS is missing')
    return "SELECT ETL.GENERATE_SEQ(\"+import_spec.parameters[\"RECORDS\"]+\" )"

def run(ctx):
    for i in range(1, ctx["num"]+1):
        ctx.emit(i)
/

IMPORT INTO (val int) FROM SCRIPT etl.generate_seq WITH records='5';

VAL
-----
1
2
3
4
5
```

Within the UDF script `generate_seq`, the method `generate_sql_for_import_spec(import_spec)` is implemented that creates a pretty simple SELECT statement and forwards the parameter RECORDS. Further, within the same UDF script a `run()` method is implemented that handles the actual logic of the data generation. In general, the UDF script specified in the IMPORT command could just create the SQL text (`generate_sql_for_import_spec(import_spec)`) and call a different UDF script. Please keep in mind that the call of UDF scripts within the SELECT has to be schema-qualified.

The example also shows how the IMPORT parameter (see WITH clause of the [IMPORT](#) command) is handed over to the script call within the SELECT. The object `import_spec` provides the script various information, e.g. parameters (WITH clause), the used connection or the column names of the target table of the IMPORT statement.

By using this information, you can dynamically control the appropriate SQL text generation and the interaction with the external system. The specific name of the metadata object and its content can vary across the script languages and is specified in detail in [Section 3.6.3, “Details for different languages”](#).

For more details, we recommend to have a look at our open source project "Hadoop ETL UDFs" which implements the data transfer to/from Hadoop systems via IMPORT/EXPORT commands by using UDF scripts.: <https://www.github.com/exasol/hadoop-etl-udfs>

The IMPORT FROM SCRIPT statement has several advantages:

- IMPORT remains the central command for loading data into Exasol
- The user interface is simple and intuitive. Just define the parameters required for your IMPORT. This hides the complexity of the SQL statements, which can become complex for real-world scenarios.
- IMPORT FROM SCRIPT supports named parameters, which makes it easier to work with mandatory and optional parameters and improves the readability.
- Similar to the normal IMPORT, the IMPORT FROM SCRIPT command can be used as subselect (and hence in views) which allows you to embed the data access.

3.4.5. User-defined EXPORT using UDFs



Please note that this feature has been added as beta-feature afterwards to version 6.0
Therefore, the API may change in the future and could differ slightly to this description.

Similar to the concept of user-defined IMPORT processes, you can implement user-defined EXPORT processes. You simply have to create a UDF script that establishes a connection to the external system within the script code and transfers the data afterwards. A simple SELECT my_export_script(. . .) FROM <table> could send all table rows to an external system. It is recommended to use a SET script and control the parallelism via a GROUP BY clause to minimize the overhead of establishing connections.

```
SELECT my_export_script('<service_address>', filepath)
FROM my_table GROUP BY partition;
```

EXPORT INTO SCRIPT Syntax

Just like the IMPORT command, UDF scripts can be easily used via the EXPORT INTO SCRIPT syntax (for details about the syntax please see [EXPORT](#)).

The only precondition is again that the UDF script implements a specific method that creates a corresponding SQL string using the export information. Specifics about this method for the various script languages can be found in [Section 3.6.3, “Details for different languages”](#).

In the following section you can find a corresponding example for our open-sourced Hadoop integration.

3.4.6. Hadoop and other systems

The technology of user-defined ETL processes using UDF scripts and the integrated IMPORT/EXPORT SCRIPT syntax can be easily leveraged to integrate diverse external systems into Exasol. Especially due to the flexibility and the power of the script languages you can quickly adjust the UDF scripts to specific new data formats or individual requirements of your systems.

In our open source repository (see <https://github.com/exasol>) we provide UDF scripts for easily integrating external systems. Just download the corresponding UDF scripts, execute them on your Exasol database, and you can already start in most cases. And we would be very happy if you would contribute to our open source community by using, extending and adding to our open-source tools.

One of the existing script implementations is an integration for Hadoop systems (see <https://www.github.com/exasol/hadoop-etl-udfs>). In the following example, Apache HCatalog™ tables are imported and exported using the provided UDF scripts import_hcat_table and export_hcat_table. You can see how simple the resulting commands look like.

```

IMPORT INTO my_table_1 FROM SCRIPT etl.import_hcat_table
WITH
  HCAT_DB      = 'default'
  HCAT_TABLE   = 'my_hcat_table_1'
  HCAT_ADDRESS = 'hcatalog-server:50111'
  HDFS_USER    = 'hdfs';

EXPORT my_table_2 INTO SCRIPT etl.export_hcat_table
WITH
  HCAT_DB      = 'default'
  HCAT_TABLE   = 'my_hcat_table_2'
  HCAT_ADDRESS = 'hcatalog-server:50111'
  HDFS_USER    = 'hdfs';

```

3.4.7. Using virtual schemas for ETL

Virtual schemas can be quite handy for implementing a simple loading process. Once a virtual schema is defined and the tables of the external data source visible, an ETL job can very easily materialize these tables using just the simple statement `CREATE TABLE materialized_schema.my_table AS SELECT * FROM virtual_schema.my_table`. You can even run complex transformations directly in the subselect.



If you are not familiar with the concept of virtual schemas, we refer to [Section 3.7, “Virtual schemas”](#).

If you don't want to materialize all tables, you could also create a mix of views directing to the virtual tables and materialized ones and decide case by case which tables should be permanently stored within Exasol to gain optimal performance.

A further advantage of virtual tables is that you don't need to adjust the ETL process if columns were introduced or renamed in the source system.

3.4.8. Definition of file formats (CSV/FBV)

The CSV Data format

Since [CSV](#) is not standardized, the format supported by Exasol is defined here.

The following formatting rules for data records are to be observed:

Comments

A data record which has a "#" (sharp) as its first character is ignored.
Comments, header and footer can be used in this manner.

Separation of the fields

The fields within a data record are separated with the field separator.

John, Doe,120 Any St., Anytown

Row separation

The data records are separated from one another with the row separator.
Each row is equal to one data record.

John, Doe,120 any str., Anytown, 12a John, Doe,120 any str., Anytown, 12b
--

Within a field the following rules apply for data presentation:

Spacing characters Spacing character at the beginning and/or the end of a field can optionally be trimmed by specifying the TRIM option in the **IMPORT** statement. In case of the option RTRIM, the data

```
John , Doe ,120 Any St.
```

will be interpreted as:

```
John, Doe,120 Any St.
```

NULL value

NULL is represented as an empty data field.

Hence,

```
John,,Any St.
```

will be interpreted as:

```
John,NULL,Any St.
```



Please consider that inside the database, empty strings are interpreted as NULL values.

Numbers

Numbers can be used in floating point or exponential notation. Optionally you can use a format (see also [Section 2.6.2, “Numeric format models”](#)). Please also consider the settings for the session parameter **NLS_NUMERIC_CHARACTERS**.

Examples:

```
John, Doe,120 Any St.,123,12.3,-1.23,-1.21E+10,4.1E-12
```

Timestamp/Date

If no explicit format (see also [Section 2.6.1, “Date/Time format models”](#)) was specified, then the current default formats are used, defined by the session parameters **NLS_DATE_FORMAT** and **NLS_TIMESTAMP_FORMAT**. Please also consider the session parameter **NLS_DATE_LANGUAGE** for certain format elements.

Boolean value

Boolean values are TRUE or FALSE. Those values can be represented by different values like e.g. 0 for FALSE or T for TRUE (see also [Boolean data type](#)).

Special characters

If the field separator, field delimiter or row separator occurs within the data the affected field must be enclosed in the field delimiter. To contain the field delimiter itself in the data it has to be written twice consecutive.

Example (field separator: comma, field delimiter: double quotes):

```
Conference room 1,"John, " "please"" call me back! ", "
```

will be read as a data record with three fields:

Field #1: Conference room 1

Field #2: John, "please" call me back!

Field #3: Empty string which corresponds to a NULL value within the database!

The same example without using the field delimiter will result in an error:

```
Conference room 1,John, "please" call me back! , "
```

The Fixblock Data format (FBV)

Alongside the [CSV](#) data format, Exasol also supports text files with a fixed byte structure - the so-called [FBV](#) format (Fix Block Values). These have a fixed width per column, as a result they can be parsed more quickly. A delimiter between the individual fields is not necessary.

For the interpretation of the data you have to consider the following elements:

Column width	A fixed number of bytes is used per column. If the size of the content is smaller than the column width, the content is supplemented with the specified padding characters.
Column alignment	With left alignment the content is at the beginning of the column and the padding characters follow. With right alignment the content is at the end of the column and is preceded by the padding characters.
Row separation	The linefeed character optionally follows at the end of a data record.
NULL values	In order to realize NULL values , a column is written across the entire length with padding characters.
Numbers	Numbers are stored in a form readable by humans, whereby you can import floating point and scientific notation numbers. However, the column width must be maintained and padding characters used as necessary.
Explicit formats	Optionally you can specify a format for numbers or datetime values. Please consider Section 2.6.2, “Numeric format models” and Section 2.6.1, “Date/Time format models” .

3.5. Scripting

3.5.1. Introduction

Comparison between scripting programs and UDF scripts

Chapter [Section 3.6, “UDF scripts”](#) describes a programming interface with which customers can develop individual analyses. These user defined functions or scripts are available to flexibly compute big amounts of data within a SELECT command.

In this chapter we introduce the scripting programming, which is in contrary an interface for executing several SQL commands sequentially and for handling errors during those executions. Hence you can run a control jobs within the database (e.g. complex loading processes) and ease up repeating jobs by parameterized scripts - like the creation of an user including its password and privileges.

Additionally, you can indeed process result tables of SQL queries, but a scripting program is a sequential program and only runs on a single cluster node (except the contained SQL statements). Therefore it is not reasonable to do iterative operations on big data sets. For this purpose we recommend the use of user defined functions (see [Section 3.6, “UDF scripts”](#)).

-  • For control jobs including several SQL commands you can use scripting programs
- Iterative operations on big data sets can be done via user defined functions (see also [Section 3.6, “UDF scripts”](#)).

For scripting programs, only the programming language *Lua* is available (in version 5.1), extended by a couple of specific features. The illustration of the Lua language in the following chapters should enable you to work with the scripting language of Exasol. But if you want to learn the fundamentals of Lua, we recommend to read the official documentation (see also <http://www.lua.org>).

A script program is created, executed and dropped by the commands `CREATE SCRIPT`, `EXECUTE SCRIPT` and `DROP SCRIPT`. The return value of the `EXECUTE SCRIPT` command is either a number (as rowcount) or a result table.

Example

In this example a script is created which can simply copy all tables from one schema into a new schema.

 The ending slash (/) is only required when using EXAplus.

```
CREATE SCRIPT copy_schema (src_schema, dst_schema) AS
  -- define function if anything goes wrong
  function cleanup()
    query([[DROP SCHEMA ::s CASCADE]], {s=dst_schema})
    exit()
  end

  -- first create new schema
  query([[CREATE SCHEMA ::s]], {s=dst_schema})

  -- then get all tables of source schema
  local success, res = pquery([[SELECT table_name FROM EXA_ALL_TABLES
                                WHERE table_schema=:s]], {s=src_schema})
  if not success then
```

```

cleanup()
end

-- now copy all tables of source schema into destination
for i=1, #res do
    local table_name = res[i][1]
    -- create table identifiers
    local dst_table = join(".", quote(dst_schema), quote(table_name))
    local src_table = join(".", quote(src_schema), quote(table_name))
    -- use protected call of SQL execution including parameters
    local success = pquery([[CREATE TABLE ::d AS SELECT * FROM ::s]],
                           {d=dst_table, s=src_table})
    if not success then
        cleanup()
    end
end
/

EXECUTE SCRIPT copy_schema ('MY_SCHEMA', 'NEW_SCHEMA');

```

3.5.2. General script language

Lexical conventions

Contrary to the general SQL language, the script language is case-sensitive, that means upper and lower case has to be considered for e.g. variable definitions. Furthermore, there exists a constraint that variable and function identifiers can only consist of ASCII characters. However, the ending semicolon (;) after a script statement is optional.



Scripts are case-sensitive, thus also for variable and function names



Variable- and function identifiers may only consist of ASCII characters

Comments

There are two different kinds of comment in Exasol:

- Line comments begin with the character -- and indicate that the remaining part of the current line is a comment.
- Block comments are indicated by the characters /* and */ and can be spread across several lines. All of the characters between the delimiters are ignored.

Examples

```

-- This is a single line comment

/*
  This is
  a multiline comment
*/

```

Types & Values

The following types are distinguished within the script language:

Type	Range of values
nil	nil is the "unknown type"
null and NULL	null and NULL represent the SQL NULL. This constant is not included in the Lua language and was added by us to allow comparisons with result data and returning NULL values.
boolean	Boolean values (true, false)
string	String values are specified in single or double quotes ('my_string', "my_string") and consist of any 8 bit characters. Alternatively, you can enclose string values in double square brackets ([[my_string]]). This notation is especially useful if quotes shall be used in the string and if you don't want to use escape characters ('SELECT * FROM "t" WHERE v='abc\';' equates to "SELECT * FROM \"t\" WHERE v='abc';" or simply [[SELECT * FROM "t" WHERE v='abc';]]).
number	Integers or floating point numbers (e.g. 3, 3.1416, 314.16e-2, 0.31416E1)
decimal	Decimal values (e.g. 3, 3.1416)

Please note that the type decimal is not a standard Lua type, but a user-defined Exasol type (of type userdata, similar to the special value NULL). This decimal type supports the following operators and methods for the usual mathematical calculations and conversions:

Constructor	
decimal(value [,precision [, value can be of type string, number or decimal. scale]])	 The default for precision and scale is (18,0), i.e. decimal(5.1) is rounded to the value 5!
Operators	
+, -, *, /, %	Addition, subtraction, multiplication, division and modulo calculation of two numerical values. The return type is determined dynamically: decimal or number
== , <, <=, >, >=	Comparison operators for numerical values. Return type: boolean
Methods	
var:add(), var:sub(), var:mul(), var:mod()	Addition, subtraction, multiplication and modulo calculation of two numerical values.  No new variable is created in this case!
var:scale(), var:prec()	Scale and precision of a decimal. Return type: number
var:tonumber()	Conversion into a number value. Return type: number
var:tostring(), tostring(var)	Conversion into a string value. Return type: string

In the following you find some examples of using decimal values:

```
d1 = decimal(10)
d2 = decimal(5.9, 2, 1)
s = d1:scale()      -- s=0
str = tostring(d2)   -- str='5.9'
d1:add(d2)         -- result is 16
```

Simple Variables

Script variables are typed dynamically. That means that variables have no type, but the values which are assigned to the variables. An assignment is done by the operator =.

As default, the scope of a variable is global. But it can be limited to the current execution block by using the keyword local.



We recommend to use local variables to explicitly show the variable declaration.

Examples

```
local a = nil      -- nil
local b = false    -- boolean
local c = 1.0      -- number
local d = 'xyz'    -- string
d = 3              -- same variable can be used for different types
local e,f = 1,2    -- two variable assignments at once
g = 0              -- global variable
```

Arrays

An array consists of a list of values (`my_array={2,3,1}`) which can also be heterogeneous (with different types).

An element of an array can be accessed through its position, beginning from 1 (`my_array[position]`). The size of an array can be determined by the # -operator (`#my_array`). In case of a nil value you will get an exception.

The elements of an array can also be an array. That is how you can easily create multidimensional arrays.

Examples

```
local my_array = {'xyz', 1, false}      -- array
local first     = my_array[1]           -- accessing first entry
local size      = #my_array            -- size=3
```

Dictionary Tables

Beside simple variables and arrays you can also use dictionary tables which consist of a collection of key/value pairs. The keys and values can be heterogeneous (with different types).

The access to a specific value is accomplished by the key, either by using the array notation (`variable[key]`) or by using the point notation (`variable.key`).

By using the function `pairs(t)` you can easily iterate through all entries of the dictionary (`for k,v in pairs(t) do end`).



In the Lua documentation there is no difference between arrays and dictionary tables - they are both simply named *table*.

Examples

```

local my_contact = {name='support',           -- define key/value pairs
                    phone='unknown'}
local n = my_contact['phone']                -- access method 1
n = my_contact.phone                         -- access method 2
my_contact.phone = '0049911239910'           -- setting single value

-- listing all entries in dictionary
for n,p in pairs(my_contact) do
    output(n..":"..p)
end

-- defining a 2 dimensional table
local my_cube = {{10, 11, 12}, {10.99, 6.99, 100.00}}
local my_value = my_cube[2][3]                 -- -> 100.00

-- defining "column names" for the table
local my_prices = {product_id={10, 11, 12}, price={10.99, 6.99, 100.00}}
local my_product_position = 3
local my_product_id = my_prices.price[my_product_position] -- -> 100.00
local my_price = my_prices.product_id[my_product_position] -- -> 12

```

Execution blocks

Execution blocks are elements which limit the scope of local variables. The script itself is the outermost execution block. Other blocks are defined through control structures (see next section) or function declarations (see section [Functions](#)).

Explicit blocks can be declared via `do end`. They are mainly useful to limit the scope of local variables.

Examples

```

-- this is the outermost block
a = 1           -- global variable visible everywhere

if var == false then
    -- the if construct declares a new block
    local b = 2   -- b is only visible inside the if block
    c = 3         -- global visible variable
end

-- explicit declared block
do
    local d = 4; -- not visible outside this block
end

```

Control structures

The following control structures are supported:

Element	Syntax
if	<code>if <condition> then <block></code> <code>[elseif <condition> then <block>]</code>

Element	Syntax
	[else <block>] end
while	while <condition> do <block> end
repeat	repeat <block> until <condition>
for	for <var>=<start>,<end>[,<step>] do <block> end
	for <var> in <expr> do <block> end

Notes

- The condition `<condition>` is evaluated as `false` if its value is `false` or `nil`, otherwise it is evaluated as `true`. That means in particular that the value `0` and an empty string is evaluated as `true`!
- The control expressions `<start>`, `<end>`, and `<step>` of the `for` loop are evaluated only once, before the loop starts. They must all result in numbers. Within the loop, you may not assign a value to the loop variable `<var>`. If you do not specify a value for `<step>`, then the loop variable is incremented by `1`.
- The `break` statement can be used to terminate the execution of a `while`, `repeat` and `for`, skipping to the next statement after the loop. For syntactic reasons, the `break` statement can only be written as the last statement of a block. If it is really necessary to break in the middle of a block, then an explicit block can be used (`do break end`).

Examples

```

if var == false
  then a = 1
  else a = 2
end

while a <= 6 do
  p = p*2
  a = a+1
end

repeat
  p = p*2
  b = b+1
until b == 6

for i=1,6 do
  if p< 0 then break end
  p = p*2
end

```

Operators

The following operators are supported withing the script language:

Operators	Notes
+, -, *, /, %	Common arithmetic operators ⚠ Please note that float arithmetic is always used.
^	Power ($2^3=8$)
==, ~=	If the operands of the equality operator (==) are different, the condition is always evaluated as <code>false</code> ! The inequality operator (~=) is exactly the opposite of the equality operator.
<, <=, >, >=	Comparison operators
and, or, not	<ul style="list-style-type: none"> and returns the first operand, if it is <code>nil</code> or <code>false</code>, otherwise the second one or returns the first operand, if it is not <code>nil</code> or <code>false</code>, otherwise the second one Both operators use short-cut evaluation, that is, the second operand is evaluated only if necessary not only returns <code>true</code>, if the operand is <code>nil</code> or <code>false</code>, otherwise it returns <code>false</code>.
..	Concatenation operator for strings and numerical values

Operator precedence follows the order below, from higher to lower priority:

1. ^
2. not, - (unary operand)
3. *, /, %
4. +, -
5. ..
6. <, >, <=, >=, ~=, ==
7. and
8. or

You can use parentheses to change the precedences in an expression.

Examples

```
local x = 1+5      --> 6
x = 2^5          --> 32

x = 1==1         --> true
x = 1=='1'        --> false
x = 1~='1'        --> true

x = true and 10   --> 10
x = true and nil  --> nil
x = 10 and 20    --> 20
x = false and nil --> false
x = nil and false --> nil

x = true or false --> true
x = 10 or 20     --> 10
x = false or nil  --> nil
x = nil or false  --> false
```

```
x = nil    or 'a'      --> 'a'

x = not true        --> false
x = not false       --> true
x = not nil         --> true
x = not 10          --> false
x = not 0           --> false

x = 'abc'..'def'    --> 'abcdef'
```

Functions

Scripts can be easily structured by the usage of functions.

Syntax

```
function <name> ( [parameter-list] )
  <block>
end
```

Notes

- Simple variables are treated as *per value* parameters. That means they cannot be manipulated within the function. However, arrays and dictionary tables are treated as *per reference* parameters which means their entries are mutable. But if you assign a complete new object, then the original function parameter is not affected.
- If you call a function with too many arguments, the supernumerous ones are ignored. If you call it with too few arguments, the rest of them are initialized with `nil`.
- Via `return`, you can exit a function and return one or more return values. For syntactic reasons, the `return` statement can only be written as the last statement of a block, if it returns a value. If it is really necessary to return in the middle of a block, then an explicit block can be used (`do return end`).
- Functions are first-class values, they can be stored in a variable or passed in a parameter list.

Examples

```
function min_max(a,b,c)
  local min,max=a,b
  if a>b then min,max=b,a
  end
  if c>max then max=c
  elseif c<min then min=c
  end
  return min,max
end

local lowest, highest = min_max(3,4,1)
```

ERROR Handling via `pcall()` and `error()`

Normally, a script terminates whenever any error occurs. However in some cases you need to handle special errors and want to do some actions. For that reason, there exist some functions for that purpose:

`pcall()` Can be used to protect a function call. The parameters are the function name and all parameters of the function, e.g. `pcall(my_function,param1,param2)` instead of `my_function(param1,param2)`. The function name `pcall` stands for *protected call*.

The function `pcall()` returns two values:

1. Success of the execution: `false` if any error occurred
2. Result: the actual result if no error occurred, otherwise the exception text

These return values can be assigned to variables and evaluated afterwards (e.g. `success, result=pcall(my_function, param1)`).

`error()` Throws an error which terminates a function or the script.

Examples

```
-- define a function which can throw an error
function divide(v1, v2)
    if v2==0 then
        error()
    else
        return v1/v2
    end
end

-- this command will not abort the script
local success, result=pcall(divide, 1, 0)
if not success then
    result = 0
end
```

3.5.3. Database interaction

Executing SQL statements via `query()` and `pquery()`

To execute SQL commands, the two functions `query()` and `pquery()` can be used which accept a string value as input parameter. Especially for those string values, the alternative string notation (`[[SELECT ...]]`) is useful to avoid problems with single or double quotes within the SQL commands. Even though the ending semicolon after SQL commands is mandatory in EXAplus, it is optional within database scripts.

If an error occurs during the call of the `query()`, the script terminates and returns the corresponding exception message. To protect such a call, you can use either the special function `pquery()` or `pcall()` (see also section [ERROR Handling via `pcall\(\)` and `error\(\)`](#)).

Return values of function `query()`

The return values of `query()` delivers diverse information which depends of the query type:

1. SELECT statements

Returns a two dimensional array whose values are *read-only*. If you want to process these values, you have to create another two-dimensional array and copy the values. However, please consider that scripts are not applicable for operations on big data volumes.

The first dimension of the result represents the rows, the second one the columns. Beside addressing by numbers you can also use column names, though you have to consider the case (e.g. `result[i]["MY_COLUMN"]`).

The number of rows can be determined via the `#` operator (`#result`), the number of columns via the `#` operator on the first row (`#result[1]`).

Additionally, you can access the SQL text via the key `statement_text`.

2. Other statements

Returns a dictionary table containing the following keys:

- `rows_inserted`: Number of inserted rows
- `rows_updated`: Number of updated rows
- `rows_deleted`: Number of deleted rows
- `rows_affected`: Sum of the other three values
- `statement_text`: SQL text of the executed statement
- `statement_id`: Statement ID within the session as string value

When executing `IMPORT` and `EXPORT` commands, the following keys are defined:

- `etl_rows_written`: number of inserted rows
- `etl_rows_with_error`: number of invalid rows (only reasonable if the REJECT LIMIT option was used)
- `etl_rows_read`: number of read rows (sum of two numbers above)

Return values of function `pquery()`

Similar to `pcall()`, the `pquery()` function returns two values ():

1. Success of the execution: `false` in case of any error
2. The actual result of the query, if no error occurred. Otherwise a dictionary table containing the keys `error_message` and `error_code`. In both cases, you can access the SQL text via the key `statement_text`.



The result of the functions `query()` and `pquery()` is read-only.

Parametrized SQL commands

By using parametrized SQL commands, you can execute dynamic SQL statements without having to create new SQL text. You could e.g. execute a command within a script to create a table multiple times by passing different table names. In such a case, you can use the table name as parameter of the SQL statement.

However, such parameters are evaluated either as identifier or as value before they are placed into the text. The reason for that approach is to avoid any security risks through *SQL Injection*.

The following kinds of parameters can be used within SQL statements:

- `:variable` Evaluates the content of the variable as constant value, e.g. in a column filter (`query([[SELECT * FROM t WHERE i=:v]] , {v=1})`).
- `::variable` Evaluates the content of the variable as identifier. Delimited identifier have to be delimited by double quotes withing the string (e.g. `table_name=["t"]`).

If you use a parametrized `query()` or `pquery()` function, you have to specify a dictionary table as second function parameter to define the variables. In this definition you can either specify constant values (e.g. `query([[SELECT * FROM t WHERE ::c=:v]] , {c=column_name,v=1})`) or assign script variables. It is not possible to directly use script variables withing the functions `query()` and `pquery()`.

In case no second parameter is specified, the SQL text is executed without interpretation. This is e.g. important if you want to create a script via `query()` or `pquery()` which contain such variables in the text.

Examples

```

/*
 *  just executing simple SQL statements
 */
query([[CREATE TABLE t(c CHAR(1))]])      -- creates table T
query([[CREATE TABLE "t"(c CHAR(1))]])    -- creates table t
local my_sql_text = [[INSERT INTO "T" VALUES 'a','b','c','d']]
query(my_sql_text)

/*
 *  using result tables of a query by concatenating
 *  all table entries into one result string
 */
local res_table = query([[SELECT c FROM t]])
local first_element = res_table[1][1]      -- access first row of first column
local row_size = #res_table                -- row_size=4
if row_size==0 then
    col_size = 0
else
    col_size=#res_table[1]                 -- col_size=1
end
local concat_str = ''
for col=1, col_size do
    for row=1, row_size do
        concat_str = concat_str..res_table[row][col]  --> 'abcd'
    end
end

/*
 *  using result information about manipulated rows
 */
local res = query([
    MERGE INTO staff s USING update_table u ON (s.id=u.id)
    WHEN MATCHED THEN UPDATE SET s.salary=u.salary DELETE WHERE u.delete_flag
    WHEN NOT MATCHED THEN INSERT VALUES (u.id,u.salary)
])
local i, u, d = res.rows_inserted, res.rows_updated, res.rows_deleted

/*
 *  using pquery to avoid abort of script
 */
local success, result = pquery([[DROP USER my_user]])
if not success then
    -- accessing error message
    local error_txt = 'Could not drop user. Error: '..result.error_message
    query([[INSERT INTO my_errors VALUES (CURRENT_TIMESTAMP, :err)]],
          {err=error_txt})
end

/*
 *  now using variables inside the SQL statements
 *  and create 5 tables at once
*/

```

```

*/  

for i=1,5 do  

    local tmp_table_name = 'TABLE_'.i  

    query([[CREATE TABLE ::t (c CHAR(1))]], {t=tmp_table_name})  

end  

-- distribute 3 values to the first 3 tables created before  

local values = {'x', 'y', 'z'}  

for i=1,#values do  

    local table_name='TABLE_'.i  

    query([[INSERT INTO ::t VALUES :v]], {t=table_name,v=values[i]})  

end  

-- using both types of variables at once  

query([[SELECT * FROM t WHERE ::x=:y]],{x='c',y='a'})
```

Scripting parameters

To be able to pass parameters to a script, you can specify them simply as an input list within the script definition ([CREATE SCRIPT](#)). Those parameters do not need any type definition. Simple variables are just specified as identifiers which are case-sensitive like within the script.

If you want to pass an array as parameter, you have to declare that by using the keyword ARRAY before the identifier. When calling a script, you have to use the construct ARRAY(value1, value2, ...). By using this concept, you can easily pass dynamic lists to a script. E.g. a script could be called for a list of user names and could grant a certain system privilege, independent to the number of passed users.

Examples

```

-- using script parameter for table name
create script create_single_table (table_name) as
    query([[CREATE TABLE ::t (i INT)]], {t=table_name})
/  

-- using ARRAY construct for multiple input
create script create_multiple_tables (ARRAY table_names) as
    for i=1,#table_names do
        query([[CREATE TABLE ::t (i INT)]], {t=table_names[i]})  

    end
/  

EXECUTE SCRIPT create_single_table ('t1');
EXECUTE SCRIPT create_multiple_tables (ARRAY('t2','t3','t4'));
SELECT * FROM CAT;  

TABLE_NAME TABLE_TYPE
-----  

T1      TABLE
T2      TABLE
T3      TABLE
T4      TABLE
```

IMPORT of external scripts

To be able to access functions which are defined in external scripts, you can use the `import()` function. This function executes the script for initialization reasons and provides all functions and global defined variables in a namespace which can be addressed via the script name. Alternatively, you can pass an alias as second parameter.



When including a script via the `import()` function, the imported script is executed and all functions and global variables are provided in the corresponding namespace.

In the following example the script `other_script` is created which defines the function `min_max()`. This function can be accessed in the second script `my_script` after the inclusion via `import("schema1.other_script", "my_alias")` by using the syntax `my_alias.min_max()`.

If you want to import a parametrized script, you have to specify those params in the call of the `import()` function at the end, that means after the optional alias. Such parameters can be either constant values or script variables.

Examples

```
CREATE SCRIPT schema1.other_script AS
  function min_max(a,b,c)
    local min,max=a,b
    if a>b then min,max=b,a
    end
    if c>max then max=c
    elseif c<min then min=c
    end
    return min,max
  end
/
CREATE SCRIPT schema2.my_script (param1, param2, param3) AS
import('schema1.other_script', 'my_alias')
-- accessing external function through alias
local lowest, highest = my_alias.min_max(param1, param2, param3)
output(lowest...',..highest)
/
```

Return value of a script

To return values in a script you can use the function `exit()`. This function can be located at any place in the script, terminates the script and returns the value of the specified parameter.

The possible return type is specified within the `CREATE SCRIPT` command:

1. Option RETURNS ROWCOUNT or absence of any option

The value which is returned by the drivers as *rowcount*. If you omit the `exit()` parameter or if the script is not exited explicitly, then this value is 0. If you directly return the result of a query execution via `query()` or `pquery()` (e.g. `exit(query([[INSERT INTO...]]))`), then the rowcount of that query is passed (not possible when executing a `SELECT` query). Alternatively, you can specify the rowcount explicitly by passing a dictionary table which contains the key `rows_affected` (e.g. `exit({rows_affected=30})`).

2. Option RETURNS TABLE

If you specify this option the result of the script execution will be a table.

If no parameter is passed to the `exit()` function or if the script is not terminated explicitly, the result is an empty table. If you directly return the result of a query execution via `query()` or `pquery()` (e.g. `exit(query([[SELECT...]]))`), then the result table of this query is passed (only possible for SELECT queries).

Alternatively, you can specify a two dimensional array. In that case you however have to specify the column types of the result table as second parameter (see example - analog to the column definition of a CREATE TABLE command).



The result table of a script can be processed in other scripts. But a persistent storage of that table in the database via SQL is not possible (like to `CREATE TABLE <table> AS SELECT...`).



If you specify the `WITH OUTPUT` option within the `EXECUTE SCRIPT` statement, the `RETURNS` option of the script creation is overwritten (see section [Debug output](#)).

Examples

```
-- script which returns rowcount
CREATE SCRIPT script_1 AS
    function my_function(a, b)
        if a == true then
            exit()          -- exit whole script
        else
            return b, b+1 -- returning result of function
        end
    end
    local x, y = my_function(false, 5) --> y=6
    exit({rows_affected=y}) -- return rows_affected
/

-- script which returns result table of query
CREATE SCRIPT script_2 RETURNS TABLE AS
    exit(query([[SELECT * FROM DUAL]]))
/


-- return explicitly created table
CREATE SCRIPT script_3 RETURNS TABLE AS
    local result_table = {{decimal(1),"abc",true},
                          {decimal(2),"xyz",false},
                          {decimal(3),nil,nil}}
    exit(result_table, "i int, c char(3), b bool")
/
EXECUTE SCRIPT script_3;

I          C      B
-----
1 abc  TRUE
2 xyz FALSE
3
```

Metadata

Within every script you can access various metadata via global variables to enhance the control of a script.

exa.meta.database_name	Database name
exa.meta.database_version	Database version
exa.meta.script_language	Name and version of the script language
exa.meta.script_name	Name of the script
exa.meta.script_schema	Schema of the script
exa.meta.script_code	Code of the script
exa.meta.current_user	Similar to function CURRENT_USER
exa.meta.session_id	Session ID
exa.meta.statement_id	Statement ID of the EXECUTE SCRIPT command within the session. Please be aware that imported scripts or executed scripts via pquery() have a different id.
exa.meta.node_count	Number of cluster nodes
exa.meta.node_id	Local node ID starting with 0
exa.meta.vm_id	Constantly 0 since there is only one single virtual machine.

Debug output

Especially during the development of scripts it is very useful to analyze the sequence of actions via debug output. That is why the `WITH OUTPUT` option of the [EXECUTE SCRIPT](#) statement is provided. If you specify this option, every output which was created via the `output()` function is returned as a result table with one column and multiple rows - independent of the actual return value of the script.

The input parameter of the `output()` function is a single string. If you pass a different value, an implicit conversion is tried, but if that conversion is not possible, the value `NULL` is inserted in the result table.

The column name of the result table is called `OUTPUT`, the column type is a `VARCHAR` with the length of the longest string which was inserted via `output()`.

If you omit the `WITH OUTPUT` option when executing a script, all debug output via `output()` is ignored.

Examples

```
CREATE SCRIPT my_script (param1, param2) AS
    output('SCRIPT started')

    if param1==false then
        output('PARAM1 is false, exit SCRIPT')
        exit()
    else
        output('PARAM2 is '..param2)
    end
    output('End of SCRIPT reached')
/

EXECUTE SCRIPT my_script (TRUE, 5) WITH OUTPUT;

OUTPUT
-----
SCRIPT started
PARAM2 is 5
End of SCRIPT reached
```

Auxiliary functions for identifiers

To simplify the work with database scripts, some auxiliary functions are provided:

- **quote(param)**

Adds quotes around the parameter and doubles embedded quotes. Primarily, this function is useful for delimited identifiers within SQL commands.

- **join(p1, p2, p3, ...)**

Generates a string which includes all parameters p2,p3,..., separated by p1. By using the function call `join(".", . . .)`, you can easily create schema qualified identifiers (see also [Section 2.1.2, “SQL identifier”](#)).

Examples

```
output("my_table")          -- regular identifier
output(quote("my_table"))   -- delimited identifier
output(join(".", "my_schema", "my_table", "my_column"))
output(join(".", 
            quote("my_schema"),
            quote("my_table"),
            quote("my_column")))
OUTPUT
-----
my_table
"my_table"
my_schema.my_table.my_column
"my_schema"."my_table"."my_column"
```

Further Notes

Line numbers	The script text is stored beginning from the first line after the AS keyword which is no blank line and can also be found in the corresponding system tables. That is why the following example returns an error in "line 5":
--------------	---

```
CREATE SCRIPT my_script AS
    -- this is the first line!
    function do_nothing()
        return
    end
    import(x) -- error: string expected
/
```

Scope-Schema	During the execution of a script the current schema is used as scope schema (except you change the schema explicitly within the script). Therefore you should consider that schema objects are preferably accessed schema qualified.
--------------	--

3.5.4. Libraries

String library

This library provides generic functions for string manipulation, such as finding and extracting substrings, and pattern matching. More information about patterns can be found below.

Please note that when indexing a string, the first character is at position 1. Indices are allowed to be negative and are interpreted as indexing backwards, from the end of the string. Thus, the last character is at position -1.

Functions

- **string.find(s, pattern [, init [, plain]])**

Looks for the first match of `pattern` in the string `s`. If it finds a match, then `find` returns the indices of `s` where this occurrence starts and ends; otherwise, it returns `nil`. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative. A value of true as a fourth, optional argument `plain` turns off the pattern matching facilities, so the function does a plain "find substring" operation, with no characters in `pattern` being considered "magic". Note that if `plain` is given, then `init` must be given as well. If the pattern has captures, then in a successful match the captured values are also returned, after the two indices.

```
local x, y = string.find('hello world', 'world')
--> x=7, y=11

local a, b, c, d = string.find('my street number is 54a', '(%d+)(%a+)')
--> a=21, b=23, c=54, d='a'
```

- **string.match(s, pattern [, init])**

Looks for the first match of `pattern` in the string `s`. If it finds one, then `match` returns the captures from the pattern; otherwise it returns `nil`. If `pattern` specifies no captures, then the whole match is returned. A third, optional numerical argument `init` specifies where to start the search; its default value is 1 and may be negative.

```
local x, y = string.match('my street number is 54a', '(%d+)(%a+)')
--> x='54'
--> y='a'
```

- **string.gmatch(s, pattern)**

Returns an iterator function that, each time it is called, returns the next captures from `pattern` over string `s`. If `pattern` specifies no captures, then the whole match is produced in each call.

The example collects all pairs key=value from the given string into a dictionary table:

```
local t = {}
local s = 'from=world, to=moon'
for k, v in string.gmatch(s, '(%w+)=(%w+)') do
    t[k] = v
end
--> t['from']='world'
--> t['to'] = 'moon'
```

- **string.sub(s, i [, j])**

Returns the substring of `s` that starts at `i` and continues until `j`; `i` and `j` may be negative. If `j` is absent, then it is assumed to be equal to -1 (which is the same as the string length). In particular, the call `string.sub(s, 1, j)` returns a prefix of `s` with length `j`, and `string.sub(s, -i)` returns a suffix of `s` with length `i`.

```
local x = string.sub('hello world', 3)
--> x='llo world'
```

- **string.gsub(s, pattern, repl [, n])**

Returns a copy of `s` in which all occurrences of the pattern have been replaced by a replacement string specified by `repl`. `gsub` also returns, as its second value, the total number of substitutions made. The optional last parameter `n` limits the maximum number of substitutions to occur. For instance, when `n` is 1 only the first occurrence of pattern is replaced.

The string `repl` is used for replacement. The character `%` works as an escape character: Any sequence in `repl` of the form `%n`, with `n` between 1 and 9, stands for the value of the `n`-th captured substring (see below). The sequence `%0` stands for the whole match. The sequence `%%` stands for a single `%`.

```
local x = string.gsub('hello world', '(%w+)', '%1 %1')
--> x='hello hello world world'

local x = string.gsub('hello world', '%w+', '%0 %0', 1)
--> x='hello hello world'

local x = string.gsub('hello world from Lua', '(%w+)%s*(%w+)', '%2 %1')
--> x='world hello Lua from'
```

- **string.len(s)**

Receives a string and returns its length. The empty string `''` has length 0.

```
local x = string.len('hello world')
--> x=11
```

- **string.rep(s, n)**

Returns a string that is the concatenation of `n` copies of the string `s`.

```
local x = string.rep('hello',3)
--> x='hellohellohello'
```

- **string.reverse(s)**

Returns a string that is the string `s` reversed.

```
local x = string.reverse('hello world')
--> x='dlrow olleh'
```

- **string.lower(s)**

Receives a string and returns a copy of this string with all uppercase letters changed to lowercase. All other characters are left unchanged.

```
local x = string.lower('hElLo World')
--> x='hello world'
```

- **string.upper(s)**

Receives a string and returns a copy of this string with all lowercase letters changed to uppercase. All other characters are left unchanged. Example:

```
local x = string.upper('hElLo World')
--> x='HELLO WORLD'
```

- **string.format(formatstring, e1, e2, ...)**

Returns a formatted version of its variable number of arguments following the description given in its first argument (which must be a string). The format string follows the same rules as the *printf()* family of standard C functions. The only differences are that the options/modifiers *, l, L, n, p, and h are not supported and that there is an extra option, q. The q option formats a string in a form suitable to be safely read back by the Lua interpreter. The options c, d, E, e, f, g, G, i, o, u, x, and X all expect a number as argument, whereas q and s expect a string.

Options:

%d, %i	Signed integer
%u	Unsigned integer
%f, %g, %G	Floating point
%e, %E	Scientific notation
%o	Octal integer
%x, %X	Hexadecimal integer
%c	Character
%s	String
%q	Safe string

Example:

```
local x = string.format('d=%d e=%e f=%f g=%g', 1.23, 1.23, 1.23, 1.23)
--> x='d=1 e=1.230000e+00 f=1.230000 g=1.23'
```

Pattern

Patterns are similar to regular expressions.

A *character class* is used to represent a set of characters.

The following combinations are allowed in describing a character class:

x	Represents the character x itself (if x is not one of the magic characters ^\$()%.[]*+-?)
.	(a dot) represents all characters
%a	represents all letters
%c	represents all control characters
%d	represents all digits
%l	represents all lowercase letters
%p	represents all punctuation characters
%s	represents all space characters
%u	represents all uppercase letters
%w	represents all alphanumeric characters
%x	represents all hexadecimal digits
%z	represents the character with representation 0
%x	represents the character x (where x is any non-alphanumeric character). This is the standard way to escape the magic characters. Any punctuation character (even the non magic) can be preceded by a `%^` when used to represent itself in a pattern.
[set]	represents the class which is the union of all characters in set. A range of characters may be specified by separating the end characters of the range with a `^-`. All classes %x described above may also be used as components in set. All other characters in set represent themselves. The interaction between ranges and classes is not defined. Therefore, patterns like [%a-z] or [a-%%] have no meaning.

Examples for character sets:

[%w_]	(or [_%w])	All alphanumeric characters plus the underscore
-------	------------	---

[0-7]	Represents the octal digits
-------	-----------------------------

[0-7%1%-] represents the octal digits plus the lowercase letters plus the `-' character
 [^set] represents the complement of set, where set is interpreted as above.

For all classes represented by single letters (%a, %c, etc.), the corresponding uppercase letter represents the complement of the class. For instance, %S represents all non-space characters.

A *pattern item* may be

- a single character class, which matches any single character in the class;
- a single character class followed by `*', which matches 0 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by `+', which matches 1 or more repetitions of characters in the class. These repetition items will always match the longest possible sequence;
- a single character class followed by `^-', which also matches 0 or more repetitions of characters in the class. Unlike `*', these repetition items will always match the shortest possible sequence;
- a single character class followed by `?', which matches 0 or 1 occurrence of a character in the class;
- %n, for n between 1 and 9; such item matches a substring equal to the n-th captured string (see below);
- %bxy, where x and y are two distinct characters; such item matches strings that start with x, end with y, and where the x and y are balanced. This means that, if one reads the string from left to right, counting +1 for an x and -1 for a y, the ending y is the first y where the count reaches 0. For instance, the item %b() matches expressions with balanced parentheses.

A *pattern* is a sequence of pattern items. A `^` at the beginning of a pattern anchors the match at the beginning of the subject string. A `\$` at the end of a pattern anchors the match at the end of the subject string. At other positions, `^` and `\$` have no special meaning and represent themselves.

A pattern may contain sub-patterns enclosed in parentheses; they describe *captures*. When a match succeeds, the substrings of the subject string that match captures are stored (captured) for future use. Captures are numbered according to their left parentheses. For instance, in the pattern "(a*(.)%w(%s*))", the part of the string matching "a*(.)%w(%s*)" is stored as the first capture (and therefore has number 1); the character matching "." is captured with number 2, and the part matching "%s*" has number 3.

As a special case, the empty capture () captures the current string position (a number). For instance, if we apply the pattern "()aa()" on the string "flaaap", there will be two captures: 3 and 5.

Unicode

If you want to process unicode characters, you can use the library `unicode.utf8` which contains the similar functions like the `string` library, but with unicode support. Another library called `unicode.ascii` exists which however has the same functionality like the library `string`.

XML parsing

The library `lxp` contains several features to process XML data. The official reference to those functions can be found under <http://matthewwild.co.uk/projects/luaexpat>.

The following methods are supported:

<code>lxp.new(callbacks, [separator])</code>	The parser is created by a call to the function <code>lxp.new</code> , which returns the created parser or raises a Lua error. It receives the callbacks table and optionally the parser separator character used in the namespace expanded element names.
<code>close()</code>	Closes the parser, freeing all memory used by it. A call to <code>parser:close()</code> without a previous call to <code>parser:parse()</code> could result in an error.
<code>getbase()</code>	Returns the base for resolving relative URIs.

getcallbacks()	Returns the callbacks table.
parse(s)	Parse some more of the document. The string s contains part (or perhaps all) of the document. When called without arguments the document is closed (but the parser still has to be closed). The function returns a non nil value when the parser has been successful, and when the parser finds an error it returns five results: nil, msg, line, col, and pos, which are the error message, the line number, column number and absolute position of the error in the XML document.
pos()	Returns three results: the current parsing line, column, and absolute position.
setbase(base)	Sets the base to be used for resolving relative URIs in system identifiers.
setencoding(encoding)	Set the encoding to be used by the parser. There are four built-in encodings, passed as strings: "US-ASCII", "UTF-8", "UTF-16", and "ISO-8859-1".
stop()	Abort the parser and prevent it from parsing any further through the data it was last passed. Use to halt parsing the document when an error is discovered inside a callback, for example. The parser object cannot accept more data after this call.

SQL parsing

The self developed library `sqlparsing` contains a various number of functions to process SQL statements. Details about the functions and their usage can be found in [Section 3.8, “SQL Preprocessor”](#).

Internet access

Via the library `socket` you can open http, ftp and smtp connections. More documentation can be found under <http://w3.impa.br/~diego/software/lusocket/>.

Math library

The math library is an interface to the standard C math library and provides the following functions and values:

math.abs(x)	Absolute value of x
math.acos(x)	Arc cosine of x (in radians)
math.asin(x)	Arc sine of x (in radians)
math.atan(x)	Arc tangent of x (in radians)
math.atan2(y,x)	Arc tangent of y/x (in radians), but uses the signs of both operands to find the quadrant of the result
math.ceil(x)	Smallest integer larger than or equal to x
math.cos(x)	Cosine of x (assumed to be in radians)
math.cosh(x)	Hyperbolic cosine of x
math.deg(x)	Angle x (given in radians) in degrees
math.exp(x)	Return natural exponential function of x
math.floor(x)	Largest integer smaller than or equal to x

<code>math.fmod(x,y)</code>	Modulo
<code>math.frexp(x)</code>	Returns m and n so that $x=m2^n$, n is an integer and the absolute value of m is in the range [0.5;1) (or zero if x is zero)
<code>math.huge</code>	Value <code>HUGE_VAL</code> which is larger than any other numerical value
<code>math.ldexp(m,n)</code>	Returns $m2^n$ (n should be an integer)
<code>math.log(x)</code>	Natural logarithm of x
<code>math.log10(x)</code>	Base-10 logarithm of x
<code>math.max(x, ...)</code>	Maximum of values
<code>math.min(x, ...)</code>	Minimum of values
<code>math.modf(x)</code>	Returns two number, the integral part of x and the fractional part of x
<code>math.pi</code>	Value of π
<code>math.pow(x,y)</code>	Returns value x^y
<code>math.rad(x)</code>	Angle x (given in degrees) in radians
<code>math.random([m,[n]])</code>	Interface to random generator function rand from ANSI C (no guarantees can be given for statistical properties). When called without arguments, returns a pseudo-random real number in the range [0;1). If integer m is specified, then the range is [1;m]. If called with m and n, then the range is [m;n].
<code>math.randomseed(x)</code>	Sets x as seed for random generator
<code>math.sin(x)</code>	Sine of x (assumed to be in radians)
<code>math.sinh(x)</code>	Hyperbolic sine of x
<code>math.sqrt(x)</code>	Square root of x
<code>math.tan(x)</code>	Tangent of x
<code>math.tanh(x)</code>	Hyperbolic tangent of x

Table library

This library provides generic functions for table manipulation. Most functions in the table library assume that the table represents an array or a list. For these functions, when we talk about the "length" of a table we mean the result of the length operator (`#table`).

Functions

- **table.insert(table, [pos,] value)**

Inserts element `value` at position `pos` in `table`, shifting up other elements if necessary. The default value for `pos` is `n+1`, where `n` is the length of the table, so that a call `table.insert(t,x)` inserts `x` at the end of table `t`.

- **table.remove(table [, pos])**

Removes from `table` the element at position `pos`, shifting down other elements if necessary. Returns the value of the removed element. The default value for `pos` is `n`, where `n` is the length of the table, so that a call `table.remove(t)` removes the last element of table `t`.

- **`table.concat(table [, sep [, i [, j]]])`**

Returns `table[i]..sep..table[i+1] ... sep..table[j]`. The default value for `sep` is the empty string, the default for `i` is 1, and the default for `j` is the length of the table. If `i` is greater than `j`, returns the empty string.

- **`table.sort(table [, comp])`**

Sorts table elements in a given order, in-place, from `table[1]` to `table[n]`, where `n` is the length of the table. If `comp` is given, then it must be a function that receives two table elements, and returns true when the first is less than the second (so that `not comp(a[i+1],a[i])` will be true after the sort). If `comp` is not given, then the standard Lua operator `<` is used instead. The sort algorithm is not stable; that is, elements considered equal by the given order may have their relative positions changed by the sort.

- **`table.maxn(table)`**

Returns the largest positive numerical index of the given table, or zero if the table has no positive numerical indices. (To do its job, this function does a linear traversal of the whole table.)

3.5.5. System tables

The following system tables show the existing database scripts:

- [EXA_USER_SCRIPTS](#)
- [EXA_ALL_SCRIPTS](#)
- [EXA_DB_SCRIPTS](#)

Further details can be found in [Appendix A, System tables](#).

3.6. UDF scripts

3.6.1. What are UDF scripts?

UDF scripts provides you the ability to program your own analyses, processing or generation functions and execute them in parallel inside Exasol's high performance cluster (*In Database Analytics*). Through this principle many problems can be solved very efficiently which were not possible with SQL statements. Via UDF scripts you therefore get a highly flexible interface for implementing nearly every requirement. Hence you become a **HPC** developer without the need for certain previous knowledge.



Please note that UDF scripts are part of the Advanced Edition of Exasol.

With UDF scripts you can implement the following extensions:

- Scalar functions
- Aggregate functions
- Analytical functions
- MapReduce algorithms
- User-defined ETL processes

To take advantage of the variety of UDF scripts, you only need to create a script ([CREATE SCRIPT](#), see [Section 2.2.1, “Definition of the database \(DDL\) ”](#)) and use this script afterwards within a SELECT statement. By this close embedding within SQL you can achieve ideal performance and scalability.

Exasol supports multiple programming languages (Java, Lua, Python, R) to simplify your start. Furthermore the different languages provide you different advantages due to their respective focus (e.g. statistical functions in R) and the different delivered libraries (XML parser, etc.). Thus, please note the next chapters in which the specific characteristics of each language is described.



The actual versions of the scripting languages can be listed with corresponding metadata functions.

Within the [CREATE SCRIPT](#) command, you have to define the type of input and output values. You can e.g. create a script which generates multiple result rows out of a single input row (SCALAR . . . EMITS).

Input values:	
SCALAR	The keyword SCALAR specifies that the script processes single input rows. It's code is therefore called once per input row.
SET	If you define the option SET, then the processing refers to a set of input values. Within the code, you can iterate through those values (see Section 3.6.2, “Introducing examples”).
Output values:	
RETURNS	In this case the script returns a single value.
EMITS	If the keyword EMITS was defined, the script can create (emit) multiple result rows (tuples). In case of input type SET, the EMITS result can only be used alone, thus not be combined with other expressions. However you can of course nest it through a subselect to do further processing of those intermediate results.

The data types of input and output parameters can be defined to specify the conversion between internal data types and the database SQL data types. If you don't specify them, the script has to handle that dynamically (see details and examples below).



Please note that input parameters of scripts are always treated case-sensitive, similar to the script code itself. This is different to SQL identifiers which are only treated case-sensitive when being delimited.

Scripts must contain the main function `run()`. This function is called with a parameter providing access to the input data of Exasol. If your script processes multiple input tuples (thus a SET script), you can iterate through the single tuples by the use of this parameter.

Please note the information in the following table:

Table 3.1. Additional basics for UDF scripts

Topic	Notes
Internal processing	<p>During the processing of an SELECT statement, multiple virtual machines are started for each script and node, which process the data independently.</p> <p>For scalar functions, the input rows are distributed across those virtual machines to achieve a maximal parallelism.</p> <p> In case of SET input tuples, the virtual machines are used per group (if you specify a GROUP BY clause). If no GROUP BY clause is defined, then only one group exists and therefore only one node and virtual machine can process the data.</p>
ORDER BY clause	<p>Either when creating a script (CREATE SCRIPT) or when calling it you can specify an ORDER BY clause which leads to a sorted processing of the groups of SET input data. For some algorithms, this can be reasonable. But if it is necessary for the algorithm, then you should already specify this clause during the creation to avoid wrong results due to misuse.</p>
Performance comparison between the programming languages	<p>The performance of the different languages can hardly be compared, since the specific elements of the languages can have different capacities. Thus a string processing can be faster in one language, while the XML parsing is faster in the other one.</p> <p>However, we generally recommend to use Lua if performance is the most important criteria. Due to technical reasons, Lua is integrated in Exasol in the most native way and therefore has the smallest process overhead.</p>

3.6.2. Introducing examples

In this chapter we provide some introducing Lua examples to give you a general idea about the functionality of user defined scripts. Examples for the other programming languages can be found in the later chapters explaining the details of each language.

Scalar functions

User defined scalar functions (keyword SCALAR) are the simplest case of user defined scripts, returning one scalar result value (keyword RETURNS) or several result tuples (keyword SET) for each input value (or tuple).

Please note that scripts have to implement a function `run()` in which the processing is done. This function is called during the execution and gets a kind of context as parameter (has name *data* in the examples) which is the actual interface between the script and the database.

In the following example, a script is defined which returns the maximum of two values. This is equivalent to the [CASE](#) expression `CASE WHEN x>=y THEN x WHEN x<y THEN y ELSE NULL`.

 The ending slash ('/') is only required when using EXAplus.

```

CREATE LUA SCALAR SCRIPT my_maximum (a DOUBLE, b DOUBLE)
    RETURNS DOUBLE AS
function run(ctx)
    if ctx.a == null or ctx.b == null
        then return null
    end
    if ctx.a > ctx.b
        then return ctx.a
        else return ctx.b
    end
end
/
SELECT x,y,my_maximum(x,y) FROM t;

X          Y          MY_MAXIMUM(T.X,T.Y)
-----
1          2          2
2          2          2
3          2          3

```

Aggregate and analytical functions

UDF scripts get essentially more interesting if the script processes multiple data at once. Hereby you can create any kind of aggregate or analytical functions. By defining the keyword `SET` you specify that multiple input tuples are processed. Within the `run()` method, you can iterate through this data (by the method `next()`).

Furthermore, scripts can either return a single scalar value (keyword `RETURNS`) or multiple result tuples (keyword `EMITS`).

The following example defines two scripts: the aggregate function `my_average` (simulates `AVG`) and the analytical function `my_sum` which creates three values per input row (one sequential number, the current value and the sum of the previous values). The latter one processes the input data in sorted order due to the `ORDER BY` clause.

```

CREATE LUA SET SCRIPT my_average (a DOUBLE)
    RETURNS DOUBLE AS
function run(ctx)
    if ctx.size() == 0
        then return null
    else
        local sum = 0
        repeat
            if ctx.a ~= null then
                sum = sum + ctx.a
            end
        until not ctx.next()
        return sum/ctx.size()
    end
end
/
SELECT my_average(x) FROM t;

MY_AVERAGE(T.X)
-----
7.75

```

```

CREATE LUA SET SCRIPT my_sum (a DOUBLE)
    EMITS (count DOUBLE, val DOUBLE, sum DOUBLE) AS
function run(ctx)
    local sum    = 0
    local count = 0
    repeat
        if ctx.a ~= null then
            sum = sum + ctx.a
            count = count + 1
            ctx.emit(count,ctx.a,sum)
        end
    until not ctx.next()
end
/

```

SELECT my_sum(x ORDER BY x) FROM t;

COUNT	VAL	SUM
1	4	4
2	7	11
3	9	20
4	11	31

Dynamic input and output parameters

Instead of statically defining input and output parameters, you can use the syntax (...) within CREATE_SCRIPT to create extremely flexible scripts with dynamic parameters. The same script can then be used for any input data type (e.g. a maximum function independent of the data type) and for a varying number of input columns. Similarly, if you have an EMITS script, the number of output parameters and their type can also be made dynamic.

In order to access and evaluate dynamic input parameters in UDF scripts, extract the number of input parameters and their types from the metadata and then access each parameter value by its index. For instance, in Python the number of input parameters is stored in the variable exa.meta.input_column_count. Please note the details of the different programming languages in section [Section 3.6.3, “Details for different languages”](#).

If the UDF script is defined with dynamic output parameters, the actual output parameters and their types are determined dynamically whenever the UDF is called. There are three possibilities:

1. You can specify the output parameters directly in the query after the UDF call using the EMITS keyword followed by the names and types the UDF shall output in this specific call.
2. If the UDF is used in the top level SELECT of an INSERT INTO SELECT statement, the columns of the target table are used as output parameters.
3. If neither EMITS is specified, nor INSERT INTO SELECT is used, the database tries to call the function default_output_columns() (the name varies, here for Python) which returns the output parameters dynamically, e.g. based on the input parameters. This method can be implemented by the user. See [Section 3.6.3, “Details for different languages”](#) on how to implement the callback method in each language.

In the example below you can see all three possibilities.

```

-- Define a pretty simple sampling script where the last parameter defines
-- the percentage of samples to be emitted.
CREATE PYTHON SCALAR SCRIPT sample_simple (...) EMITS (...) AS
from random import randint, seed

```

```
seed(1001)
def run(ctx):
    percent = ctx[exa.meta.input_column_count-1]
    if randint(0,100) <= percent:
        currentRow = [ctx[i] for i in range(0, exa.meta.input_column_count-1)]
        ctx.emit(*currentRow)
    /

-- This is the same UDF, but output arguments are generated automatically
-- to avoid explicit EMITS definition in SELECT.
-- In default_output_columns(), a prefix 'c' is added to the column names
-- because the input columns are autogenerated numbers
CREATE PYTHON SCALAR SCRIPT sample (...) EMITS (...) AS
from random import randint, seed
seed(1001)
def run(ctx):
    percent = ctx[exa.meta.input_column_count-1]
    if randint(0,100) <= percent:
        currentRow = [ctx[i] for i in range(0, exa.meta.input_column_count-1)]
        ctx.emit(*currentRow)
def default_output_columns():
    output_args = list()
    for i in range(0, exa.meta.input_column_count-1):
        name = exa.meta.input_columns[i].name
        type = exa.meta.input_columns[i].sql_type
        output_args.append("c" + name + " " + type)
    return str(", ".join(output_args))
/

-- Example table
ID      USER_NAME PAGE_VISITS
----- -----
 1 Alice          12
 2 Bob            4
 3 Pete           0
 4 Hans          101
 5 John          32
 6 Peter          65
 7 Graham         21
 8 Steve           4
 9 Bill           64
10 Claudia        201

-- The first UDF requires to specify the output columns via EMITS.
-- Here, 20% of rows should be extracted randomly.

SELECT sample_simple(id, user_name, page_visits, 20)
EMITS (id INT, user_name VARCHAR(100), PAGE_VISITS int)
FROM people;

ID      USER_NAME PAGE_VISITS
----- -----
 2 Bob            4
 5 John          32

-- The second UDF computes the output columns dynamically
```

```

SELECT SAMPLE(id, user_name, page_visits, 20)
  FROM people;

C0      C1      C2
-----
 2 Bob        4
 5 John       32

-- In case of INSERT INTO, the UDF uses the target types automatically
CREATE TABLE people_sample LIKE people;
INSERT INTO people_sample
  SELECT sample_simple(id, user_name, page_visits, 20) FROM people;

```

MapReduce programs

Due to its flexibility, the UDF scripts framework is able to implement any kind of analyses you can imagine. To show you its power, we list an example of a MapReduce program which calculates the frequency of single words within a text - a problem which cannot be solved with standard SQL.

In the example, the script `map_words` extracts single words out of a text and emits them. This script is integrated within a SQL query without having the need for an additional aggregation script (the typical Reduce step of MapReduce), because we can use the built-in SQL function `COUNT`. This reduces the implementation efforts since a whole bunch of built-in SQL functions are already available in Exasol. Additionally, the performance can be increased by that since the SQL execution within the built-in functions is more native.

```

CREATE LUA SCALAR SCRIPT map_words(w varchar(10000))
EMITS (words varchar(100)) AS
function run(ctx)
    local word = ctx.w
    if (word ~= null)
    then
        for i in unicode.utf8.gmatch(word, '([%w%p]+)')
        do
            ctx.emit(i)
        end
    end
end
/
SELECT words, COUNT(*) FROM
  (SELECT map_words(l_comment) FROM tpc.lineitem)
GROUP BY words ORDER BY 2 desc LIMIT 10;

WORDS          COUNT(*)
-----
the           1376964
slyly         649761
regular        619211
final          542206
carefully      535430
furiously      534054
ironic          519784
blithely        450438
even            425013
quickly         354712

```

Access to external services

Within scripts you can exchange data with external services which increases your flexibility significantly.

In the following example, a list of URLs (stored in a table) is processed, the corresponding documents are read from the webserver and finally the length of the documents is calculated. Please note that every script language provides different libraries to connect to the internet.

```
CREATE LUA SCALAR SCRIPT length_of_doc (url VARCHAR(50))
    EMITS (url VARCHAR(50), doc_length DOUBLE) AS
http = require("socket.http")
function run(ctx)
    file = http.request(ctx.url)
    if file == nil then error('Cannot open URL ' .. ctx.url) end
    ctx.emit(ctx.url, unicode.utf8.len(file))
end
/
SELECT length_of_doc(url) FROM t;

URL                      DOC_LENGTH
-----
http://en.wikipedia.org/wiki/Main_Page.htm      59960
http://en.wikipedia.org/wiki/Exasol              30007
```

User-defined ETL using UDFs

UDF scripts can also be used to implement very flexible ETL processes by defining how to extract and convert data from external data sources. We refer to [Section 3.4.4, “User-defined IMPORT using UDFs”](#) of [Section 3.4, “ETL Processes”](#) for further details and examples.

3.6.3. Details for different languages



For all script languages it is possible to set environment variables for the script execution. The corresponding syntax is `%env <variable>=<value>`, and one use case is to set the search path for making certain libraries accessible that are stored in BucketFS (see details below). Example: `%env LD_LIBRARY_PATH=/buckets/bfsdefault/hadoop_libs/native_lib`

Lua

Beside the following information you can find additional details for Lua in [Section 3.5, “Scripting”](#). Furthermore, we recommend the official documentation (see <http://www.lua.org>).

<code>run()</code> and <code>cleanup()</code> methods	<p>The method <code>run()</code> is called for each input tuple (SCALAR) or each group (SET). Its parameter is a kind of execution context and provides access to the data and the iterator in case of a SET script.</p> <p>To initialize expensive steps (e.g. opening external connections), you can write code outside the <code>run()</code> method, since this code is executed once at the beginning by each virtual machine. For deinitialization purpose, the method <code>cleanup()</code> exists which is called once for each virtual machine, at the end of the execution.</p>
Parameters	Note that the internal Lua data types and the database SQL types are not identical. Therefore casts must be done for the input and output data:

	<table> <tbody> <tr><td>DOUBLE</td><td>number</td></tr> <tr><td>DECIMAL and INTEGER</td><td>decimal</td></tr> <tr><td>BOOLEAN</td><td>boolean</td></tr> <tr><td>VARCHAR and CHAR</td><td>string</td></tr> <tr><td>Other</td><td>Not supported</td></tr> </tbody> </table> <p> Please also consider the details about Lua types in Section 3.5, "Scripting", especially for the special type decimal.</p> <p> NULL values are represented by the special constant NULL.</p>	DOUBLE	number	DECIMAL and INTEGER	decimal	BOOLEAN	boolean	VARCHAR and CHAR	string	Other	Not supported																												
DOUBLE	number																																						
DECIMAL and INTEGER	decimal																																						
BOOLEAN	boolean																																						
VARCHAR and CHAR	string																																						
Other	Not supported																																						
	<p>The input parameters can be addressed by their names, e.g. <code>ctx.my_input</code>.</p> <p>But you can also use a dynamic number of parameters via the notation <code>(...)</code>, e.g. <code>CREATE LUA SCALAR SCRIPT my_script (...)</code>. The parameters can then be accessed through an index, e.g. <code>data[1]</code>. The number of parameters and their data types (both determined during the call of the script) are part of the metadata.</p>																																						
Metadata	<p>The following metadata can be accessed via global variables:</p> <table> <tbody> <tr><td><code>exa.meta.database_name</code></td><td>Database name</td></tr> <tr><td><code>exa.meta.database_version</code></td><td>Database version</td></tr> <tr><td><code>exa.meta.script_language</code></td><td>Name and version of the script language</td></tr> <tr><td><code>exa.meta.script_name</code></td><td>Name of the script</td></tr> <tr><td><code>exa.meta.script_schema</code></td><td>Schema in which the script is stored</td></tr> <tr><td><code>exa.meta.current_schema</code></td><td>Schema which is currently opened</td></tr> <tr><td><code>exa.meta.script_code</code></td><td>Code of the script</td></tr> <tr><td><code>exa.meta.session_id</code></td><td>Session ID</td></tr> <tr><td><code>exa.meta.statement_id</code></td><td>Statement ID within the session</td></tr> <tr><td><code>exa.meta.current_user</code></td><td>Current user</td></tr> <tr><td><code>exa.meta.node_count</code></td><td>Number of cluster nodes</td></tr> <tr><td><code>exa.meta.node_id</code></td><td>Local node ID starting with 0</td></tr> <tr><td><code>exa.meta.vm_id</code></td><td>Unique ID for the local machine (the ids of the virtual machines have no relation to each other)</td></tr> <tr><td><code>exa.meta.input_type</code></td><td>Type of the input data (SCALAR or SET)</td></tr> <tr><td><code>exa.meta.input_column_count</code></td><td>Number of input columns</td></tr> <tr><td><code>exa.meta.input_columns[]</code></td><td>Array including the following information: {name, type, sql_type, precision, scale, length}</td></tr> <tr><td><code>exa.meta.output_type</code></td><td>Type of the output data (RETURN or EMITS)</td></tr> <tr><td><code>exa.meta.output_column_count</code></td><td>Number of output columns</td></tr> <tr><td><code>exa.meta.output_columns[]</code></td><td>Array including the following information: {name, type, sql_type, precision, scale, length}</td></tr> </tbody> </table>	<code>exa.meta.database_name</code>	Database name	<code>exa.meta.database_version</code>	Database version	<code>exa.meta.script_language</code>	Name and version of the script language	<code>exa.meta.script_name</code>	Name of the script	<code>exa.meta.script_schema</code>	Schema in which the script is stored	<code>exa.meta.current_schema</code>	Schema which is currently opened	<code>exa.meta.script_code</code>	Code of the script	<code>exa.meta.session_id</code>	Session ID	<code>exa.meta.statement_id</code>	Statement ID within the session	<code>exa.meta.current_user</code>	Current user	<code>exa.meta.node_count</code>	Number of cluster nodes	<code>exa.meta.node_id</code>	Local node ID starting with 0	<code>exa.meta.vm_id</code>	Unique ID for the local machine (the ids of the virtual machines have no relation to each other)	<code>exa.meta.input_type</code>	Type of the input data (SCALAR or SET)	<code>exa.meta.input_column_count</code>	Number of input columns	<code>exa.meta.input_columns[]</code>	Array including the following information: {name, type, sql_type, precision, scale, length}	<code>exa.meta.output_type</code>	Type of the output data (RETURN or EMITS)	<code>exa.meta.output_column_count</code>	Number of output columns	<code>exa.meta.output_columns[]</code>	Array including the following information: {name, type, sql_type, precision, scale, length}
<code>exa.meta.database_name</code>	Database name																																						
<code>exa.meta.database_version</code>	Database version																																						
<code>exa.meta.script_language</code>	Name and version of the script language																																						
<code>exa.meta.script_name</code>	Name of the script																																						
<code>exa.meta.script_schema</code>	Schema in which the script is stored																																						
<code>exa.meta.current_schema</code>	Schema which is currently opened																																						
<code>exa.meta.script_code</code>	Code of the script																																						
<code>exa.meta.session_id</code>	Session ID																																						
<code>exa.meta.statement_id</code>	Statement ID within the session																																						
<code>exa.meta.current_user</code>	Current user																																						
<code>exa.meta.node_count</code>	Number of cluster nodes																																						
<code>exa.meta.node_id</code>	Local node ID starting with 0																																						
<code>exa.meta.vm_id</code>	Unique ID for the local machine (the ids of the virtual machines have no relation to each other)																																						
<code>exa.meta.input_type</code>	Type of the input data (SCALAR or SET)																																						
<code>exa.meta.input_column_count</code>	Number of input columns																																						
<code>exa.meta.input_columns[]</code>	Array including the following information: {name, type, sql_type, precision, scale, length}																																						
<code>exa.meta.output_type</code>	Type of the output data (RETURN or EMITS)																																						
<code>exa.meta.output_column_count</code>	Number of output columns																																						
<code>exa.meta.output_columns[]</code>	Array including the following information: {name, type, sql_type, precision, scale, length}																																						
Data iterator, <code>next()</code> , <code>size()</code> and <code>reset()</code>	<p>For scripts having multiple input tuples per call (keyword SET), you can iterate through that data via the method <code>next()</code>, which is accessible through the context. Initially, the iterator points to the first input row. That's why a <i>repeat...until</i> loop can be ideally used to iterate through the data (see examples).</p> <p> If the input data is empty, then the <code>run()</code> method will not be called, and similar to aggregate functions the NULL value is returned as result (like for e.g. <code>SELECT MAX(x) FROM t WHERE false</code>).</p> <p>Additionally, there is a method <code>reset()</code> that resets the iterator to the first input element. Hereby you can do multiple iterations through the data, if this is necessary for your algorithm. The method <code>size()</code> returns the number of input values.</p>																																						
<code>emit()</code>	You can return multiple output tuples per call (keyword EMITS) via the method <code>emit()</code> .																																						

	The method expects as many parameters as output columns were defined. In the case of dynamic output parameters, it is handy in Lua to use an array using <code>unpack()</code> - e.g. <code>ctx.emit(unpack({1, "a"}))</code> .												
Import of other scripts	Other scripts can be imported via the method <code>exa.import()</code> . Scripting programs (see Section 3.5, “Scripting”) can also be imported, but their parameters will be ignored.												
Accessing connection definitions	<p>The data that has been specified when defining connections with CREATE CONNECTION is available in LUA UDF scripts via the method <code>exa.get_connection("connection_name")</code>. The result is a Lua table with the following fields:</p> <table> <tr> <td>type</td><td>The type of the connection definition. Currently only the type "PASSWORD" is used.</td></tr> <tr> <td>address</td><td>The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.</td></tr> <tr> <td>user</td><td>The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.</td></tr> <tr> <td>password</td><td>The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.</td></tr> </table>	type	The type of the connection definition. Currently only the type "PASSWORD" is used.	address	The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.	user	The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.	password	The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.				
type	The type of the connection definition. Currently only the type "PASSWORD" is used.												
address	The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.												
user	The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.												
password	The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.												
Auxiliary libraries	<p>In Section 3.5, “Scripting” you find details about the supported libraries of Lua. Shortly summarized, those are:</p> <table> <tr> <td>math</td><td>Mathematical calculations</td></tr> <tr> <td>table</td><td>Table operations</td></tr> <tr> <td>string</td><td>Text processing</td></tr> <tr> <td>unicode.utf8</td><td>Similar functions like in string, but supports unicode data</td></tr> <tr> <td>socket</td><td>Support for internet access (http, ftp, smtp - see also http://w3.impa.br/~diego/software/luasocket/)</td></tr> <tr> <td>lxml</td><td>XML parsing (see also http://matthewwild.co.uk/projects/luaexpat/)</td></tr> </table> <p> The modules <code>math</code>, <code>table</code>, <code>string</code> and <code>unicode.utf8</code> are already provided in the namespace; the others have to be loaded via <code>require()</code>.</p> <p> Additional Lua libraries cannot be installed by customers.</p>	math	Mathematical calculations	table	Table operations	string	Text processing	unicode.utf8	Similar functions like in string, but supports unicode data	socket	Support for internet access (http, ftp, smtp - see also http://w3.impa.br/~diego/software/luasocket/)	lxml	XML parsing (see also http://matthewwild.co.uk/projects/luaexpat/)
math	Mathematical calculations												
table	Table operations												
string	Text processing												
unicode.utf8	Similar functions like in string, but supports unicode data												
socket	Support for internet access (http, ftp, smtp - see also http://w3.impa.br/~diego/software/luasocket/)												
lxml	XML parsing (see also http://matthewwild.co.uk/projects/luaexpat/)												
Dynamic output parameters callback function <code>default_out_put_columns()</code>	<p>If the UDF script was defined with dynamic output parameters and the output parameters can not be determined (via specifying EMITS in the query or via INSERT INTO SELECT), the database calls the function <code>default_output_columns()</code> which you can implement. The expected return value is a String with the names and types of the output columns, e.g. "a int, b varchar(100)". See Dynamic input and output parameters for an explanation when this method is called including examples.</p> <p> You can access the Metadata <code>exa.meta</code> in the method, e.g. to find out the number and types of input columns.</p> <p> The method will be executed only once on a single node.</p>												
User defined import callback function <code>generate_sql_for_import_spec(import_spec)</code> (see Section 3.4.4, “User-	To support a user defined import you can implement the callback method <code>generate_sql_for_import_spec(import_spec)</code> . Please see also Dynamic input and output parameters and the <code>IMPORT</code> statement for the syntax. The parameter <code>import_spec</code> contains all information about the executed IMPORT FROM SCRIPT statement. The function has to generate and return a SELECT SQL statement which will retrieve the data to be imported. <code>import_spec</code> is a Lua table with the following fields:												

defined IMPORT using UDFs”)	parameters	Parameters specified in the IMPORT statement. For example the parameter FOO could be obtained by accessing <code>import_spec.parameters.FOO</code> . The value <code>nil</code> is returned if the accessed parameter was not specified.						
	is_subselect	This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.						
	subselect_column_names	This is only defined, if <code>is_subselect</code> is true and the user specified the target column names and types. It returns a list of strings with the names of all specified columns. The value <code>nil</code> is returned if the target columns are not specified.						
	subselect_column_types	This is only defined, if <code>is_subselect</code> is true and the user specified the target columns and names. It returns a list of strings with the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)"). The value <code>nil</code> is returned if the target columns are not specified.						
	connection_name	This returns the name of the connection, if such was specified. Otherwise it returns <code>nil</code> . The UDF script can then obtain the connection information via <code>exa.get_connection(name)</code> .						
	connection	This is only defined, if the user provided connection information. It returns a Lua table similar to such that is returned by <code>exa.get_connection(name)</code> . Returns <code>nil</code> if no connection information are specified.						
	 If a password was specified, it will be transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from the <code>connection_name</code> field). The actual connection information can be obtained via <code>exa.get_connection(name)</code> .							
User defined export callback function <code>g e n e r - a t e _ s q l _ f o r _ e x - p o r t _ s p e c()</code> (see Section 3.4.5, “User-defined EXPORT using UDFs”)	<p>To support a user defined export you can implement the callback method <code>generate_sql_for_export_spec(export_spec)</code>. Please see also Dynamic input and output parameters and the EXPORT statement for the syntax. The parameter <code>export_spec</code> contains all information about the executed EXPORT INTO SCRIPT statement. The function has to generate and return a SELECT SQL statement which will generate the data to be exported. The FROM part of that string can be a dummy table (e.g. DUAL) since the export command is aware which table should be exported. But it has to be specified to be able to compile the SQL string successfully.</p> <p><code>export_spec</code> is a Lua table with the following fields:</p> <table> <tr> <td>parameters</td> <td>Parameters specified in the EXPORT statement. For example the parameter FOO could be obtained by accessing <code>export_spec.parameters.FOO</code>. The value <code>nil</code> is returned if the accessed parameter was not specified.</td> </tr> <tr> <td>source_column_names</td> <td>List of column names of the resulting table that shall be exported.</td> </tr> <tr> <td>has_truncate</td> <td>Boolean value from the EXPORT command option that defines whether the content of the target table should be truncated before the data transfer.</td> </tr> </table>		parameters	Parameters specified in the EXPORT statement. For example the parameter FOO could be obtained by accessing <code>export_spec.parameters.FOO</code> . The value <code>nil</code> is returned if the accessed parameter was not specified.	source_column_names	List of column names of the resulting table that shall be exported.	has_truncate	Boolean value from the EXPORT command option that defines whether the content of the target table should be truncated before the data transfer.
parameters	Parameters specified in the EXPORT statement. For example the parameter FOO could be obtained by accessing <code>export_spec.parameters.FOO</code> . The value <code>nil</code> is returned if the accessed parameter was not specified.							
source_column_names	List of column names of the resulting table that shall be exported.							
has_truncate	Boolean value from the EXPORT command option that defines whether the content of the target table should be truncated before the data transfer.							

	has_replace	Boolean value from the EXPORT command option that defines whether the target table should be deleted before the data transfer.
	created_by	String value from the EXPORT command option that defines the creation text that is executed in the target system before the data transfer.
	connection_name	This returns the name of the connection, if such was specified. Otherwise it returns nil. The UDF script can then obtain the connection information via <code>exa.get_connection(name)</code> .
	connection	This is only defined, if the user provided connection information. It returns a Lua table similar to such that is returned by <code>exa.get_connection(name)</code> . Returns nil if no connection information are specified.
 If a password was specified, it will be transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from the connection_name field). The actual connection information can be obtained via <code>exa.get_connection(name)</code> .		

Example:

```
/*
This example loads from a webserver
and processes the following file goalies.xml:

<?xml version='1.0' encoding='UTF-8'?>
<users>
    <user active="1">
        <first_name>Manuel</first_name>
        <last_name>Neuer</last_name>
    </user>
    <user active="1">
        <first_name>Joe</first_name>
        <last_name>Hart</last_name>
    </user>
    <user active="0">
        <first_name>Oliver</first_name>
        <last_name>Kahn</last_name>
    </user>
</users>
*/


CREATE LUA SCALAR SCRIPT process_users(url VARCHAR(500))
EMITS (firstname VARCHAR(20), lastname VARCHAR(20)) AS
require("lxp")
http = require("socket.http")
local in_user_tag = false;
local in_first_name_tag = false;
local in_last_name_tag = false;
local current = {}
local users = {}

p = lxp.new(
```

```

{StartElement = function(p, tag, attr)
  if tag == "user" and attr.active == "1" then in_user_tag = true; end
  if tag == "first_name" then in_first_name_tag = true; end
  if tag == "last_name" then in_last_name_tag = true; end
end,
EndElement = function(p, tag)
  if tag == "user" then in_user_tag = false; end
  if tag == "first_name" then in_first_name_tag = false; end
  if tag == "last_name" then in_last_name_tag = false; end
end,
CharacterData = function(p, txt)
  if in_user_tag then
    if in_first_name_tag then current.first_name = txt; end
    if in_last_name_tag then current.last_name = txt; end
  end
  if current.first_name and current.last_name then
    users[#users+1] = current
    current = {}
  end
end
end}())

function run(ctx)
  content = http.request(ctx.url)
  p:parse(content); p:parse(); p:close();
  for i=1, #users do
    ctx.emit(users[i].first_name, users[i].last_name)
  end
end
/
SELECT process_users ('http://www.my_valid_webserver/goalies.xml')
FROM DUAL;

```

Java

Besides the following information you can find additional details for Java in the official documentation.

Java main class	<p>The default convention is that the script main class (which includes the methods <code>(run()</code> and optionally <code>init()</code> and <code>cleanup()</code>) must be named exactly like the name of the script (please consider the general rules for identifiers).</p> <p>You can also specify the script class explicitly using the keyword <code>%scriptclass</code> (e.g. <code>%scriptclass com.mycompany.MyScriptClass;</code>)</p> <p> All classes which are defined directly in the script are implicitly inside the Java package <code>package com.exasol</code>, because the statement <code>package com.exasol;</code> is implicitly added to the beginning of the script code.</p> <p> All callback functions (<code>run()</code>, <code>init()</code>, <code>cleanup()</code>, <code>getDefaultOutputColumns()</code> and <code>generateSqlForImportSpec()</code>) have to be implemented within the script main class.</p>
Using the Maven repository	The Exasol Java API for scripts is available in Maven to facilitate the development of Java code. Please add the following repository and dependency to the build configuration of your project (e.g. <code>pom.xml</code> for Maven):

```

<repositories>
  <repository>
    <id>maven.exasol.com</id>
    <url>https://maven.exasol.com/artifactory/exasol-releases</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>com.exasol</groupId>
    <artifactId>exasol-script-api</artifactId>
    <version>6.0.0</version>
  </dependency>
</dependencies>

```

	<pre> run(ExaMetadata, ExaIterator), init(ExaMetadata) and cleanup() methods </pre>	The method <code>static <Type> run(ExaMetadata, ExaIterator)</code> is called for each input tuple (SCALAR) or each group (SET). Its parameter is a metadata object (ExaMetadata) and an iterator for the data access (ExaIterator). To initialize expensive steps (e.g. opening external connections), you can define the method <code>static void init(ExaMetadata)</code> . This will be executed once at the beginning by each virtual machine. For deinitialization purposes, the method <code>static void cleanup(ExaMetadata)</code> exists which is called once for each virtual machine, at the end of execution.
Data iterator, <code>next()</code> , <code>emit()</code> , <code>size()</code> and <code>reset()</code>	<p>The following methods are provided in the class <code>ExaIterator</code>:</p> <ul style="list-style-type: none"> • <code>next()</code> • <code>emit(Object... values)</code> • <code>reset()</code> • <code>size()</code> • <code>getInteger(String name)</code> and <code>getInteger(int column)</code> • <code>getLong(String name)</code> and <code>getLong(int column)</code> • <code>getBigDecimal(String name)</code> and <code>getBigDecimal(int column)</code> • <code>getDouble(String name)</code> and <code>getDouble(int column)</code> • <code>getString(String name)</code> and <code>getString(int column)</code> • <code>getBoolean(String name)</code> and <code>getBoolean(int column)</code> • <code>getDate(String name)</code> and <code>getDate(int column)</code> • <code>getTimestamp(String name)</code> and <code>getTimestamp(int column)</code> <p>For scripts having multiple input tuples per call (keyword SET), you can iterate through that data via the method <code>next()</code>. Initially, the iterator points to the first input row. For iterating you can use a <code>while(true)</code> loop which is aborted in case <code>if (!ctx.next())</code>.</p> <p> If the input data is empty, then the <code>run()</code> method will not be called, and similar to aggregate functions the NULL value is returned as the result (e.g. <code>SELECT MAX(x) FROM t WHERE false</code>).</p> <p>Additionally, there is a <code>reset()</code> method which resets the iterator to the first input element. Hereby you can do multiple iterations through the data, if this is necessary for your algorithm.</p> <p>The method <code>size()</code> returns the number of input values.</p> <p>You can return multiple output tuples per call (keyword EMITS) via the method <code>emit()</code>. The method expects as many parameters as output columns were defined. In the case of dynamic output parameters, it is handy in Java to use an Object Array (Example: <code>iter.emit(new Object[] {1, "a"})</code>).</p>	

Parameter	Note that the internal Java data types and the database SQL types are not identical. Therefore casts must be done for the input and output data:
DECIMAL(p,0)	Integer, Long, BigDecimal
DECIMAL(p,s)	BigDecimal
DOUBLE	Double
DATE	java.sql.Date
TIMESTAMP	java.sql.Timestamp
BOOLEAN	Boolean
VARCHAR and CHAR	String
Other	Not supported
 The value null is the equivalent of the SQL NULL.	
The input data can be accessed via get() methods, e.g. ctx.getString("url").	
You can use a dynamic number of parameters via the notation (...), e.g. CREATE JAVA SCALAR SCRIPT my_script (...). The parameters can then be accessed through an index, e.g. ctx.getString(0) for the first parameter. The number of parameters and their data types (both determined during the call of the script) are part of the metadata.	
Metadata	The following metadata can be accessed via the object ExaMetadata:
String getDatabaseName()	Database name
String getDatabaseVersion()	Database version
String getScriptLanguage()	Name and version of the language
String getScriptName()	Name of the script
String getScriptSchema()	Schema in which the script is stored
String getCurrentSchema()	Schema which is currently opened
String getScriptCode()	Text of the script
String getSessionId()	Session ID
long getStatementId()	Statement ID within the session
long getCurrentUser()	Current user
long getNodeCount()	Number of nodes in the cluster
long getNodeID()	Local node i, starting with 0
String getVmId()	Unique ID for the local machine (the ids of the virtual machines have no relation to each other)
BigInteger getMemoryLimit()	Memory limit of the Java process
String getInputType()	Type of the input data (SCALAR or SET)
long getInputColumnCount()	Number of input columns
String getInputColumnName(int column)	Name of an input column
Class getInputColumnType(int column)	Type of an input column
String getInputColumnSqlType(int column)	SQL type of an input column
long getInputColumnPrecision(int column)	Precision of an input column
long getInputColumnScale(int column)	Scale of an input column
long getInputColumnLength(int column)	Length of an input column
String getOutputType()	Type of the output data (RETURNS or EMITS)

	<pre>long getOutputColumnCount() String getOutputColumnName(int column) Class<?> getOutputColumnType(int column) String getOutputColumnSqlType(int column) long getOutputColumnPrecision(int column) long getOutputColumnScale(int column) long getOutputColumnLength(int column) Class<?> importScript(String name)</pre>	Number of output columns Name of an output column Type of an output column SQL type of an output column Precision of an output column Scale of an output column Length of an output column Import of another script	
Exceptions	<p>The following Exasol-specific exceptions can be thrown:</p> <ul style="list-style-type: none"> • ExaCompilationException • ExaDataException • ExaIterationException 		
Import of other scripts	Besides the importScript() method of class ExaMetadata (see above), other scripts can be easily imported via the keyword %import (e.g. %import OTHER_SCRIPT;). Afterwards that script is accessible in the namespace (e.g. OTHER_SCRIPT.my_method()).		
Accessing connection definitions	The data that has been specified when defining connections with CREATE CONNECTION is available in Java UDF scripts via the method ExaMetadata.getConnection(String connectionName). The result is a Java object that implements the Interface com.exasol.ExaConnectionInformation which features the following methods: <ul style="list-style-type: none"> ExaConnectionInformation.ConnectionType ExaConnectionInformation.getType() String ExaConnectionInformation.getAddress() String ExaConnectionInformation.getUser() String ExaConnectionInformation.getPassword() 	The type of the connection definition. ConnectionType is an enumeration which currently only contains the entry PASSWORD. The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command. The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command. The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.	
Integrate your own JAR packages	Please see Section 3.6.5, “Expanding script languages using BucketFS”		
JVM Options	To enable the tuning of script performance depending on its memory requirements, the %jvmoption keyword can be used to specify the following Java VM options: <ul style="list-style-type: none"> • Initial heap size (-Xms) • Maximum heap size (-Xmx) • Thread stack size (-Xss) Example, %jvmoption -Xms128m -Xmx1024m -Xss512k;		

	<p>This sets the initial Java heap size to 128 MB, maximum heap size to 1024 MB, and thread stack size to 512 kB.</p> <p>Please note that if multiple values are given for a single option (e.g., resulting from the import of another script), the last value will be used.</p>																
Dynamic output parameters callback function <code>getDefaultOutputColumns(ExaMetadata exa)</code>	<p>If the UDF script was defined with dynamic output parameters and the output parameters can not be determined (via specifying EMITS in the query or via INSERT INTO SELECT), the database calls the method <code>static String getDefaultOutputColumns(ExaMetadata exa)</code> which you can implement. The expected return value is a String with the names and types of the output columns, e.g. "a int, b varchar(100)". See Dynamic input and output parameters for an explanation when this method is called including examples.</p> <p> The method will be executed only once on a single node.</p> <p> This function has to be implemented within the script main class.</p>																
User defined import callback function <code>generateSqlForImportSpec(ExaMetadata meta, ExaImportSpecification importSpec)</code> . Please see also Dynamic input and output parameters and the IMPORT for the syntax. The parameter <code>importSpec</code> contains all information about the executed IMPORT FROM SCRIPT statement. The function has to generate and return a SELECT SQL statement which will retrieve the data to be imported. “User-defined IMPORT using UDFs”	<p>To support a user defined import you can implement the callback method <code>public static String generateSqlForImportSpec(ExaMetadata meta, ExaImportSpecification importSpec)</code>. Please see also Dynamic input and output parameters and the IMPORT for the syntax. The parameter <code>importSpec</code> contains all information about the executed IMPORT FROM SCRIPT statement. The function has to generate and return a SELECT SQL statement which will retrieve the data to be imported.</p> <p><code>importSpec</code> is an object with the following fields:</p> <table> <tr> <td><code>Map<String, String> getParameters()</code></td> <td>Parameters specified in the IMPORT statement.</td> </tr> <tr> <td><code>boolean isSubselect()</code></td> <td>This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.</td> </tr> <tr> <td><code>List<String> getSubselectColumnNames()</code></td> <td>If <code>isSubselect()</code> is true and the user specified the target column names and types, this returns the names of all specified columns.</td> </tr> <tr> <td><code>List<String> getSubselectColumnTypes()</code></td> <td>If <code>isSubselect()</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").</td> </tr> <tr> <td><code>boolean hasConnectionName()</code></td> <td>This method returns true if a connection was specified. The UDF script can then obtain the connection information via <code>ExaMetadata.getConnection(name)</code>.</td> </tr> <tr> <td><code>String getConnectionName()</code></td> <td>If <code>hasConnection()</code> is true, this returns the name of the specified connection.</td> </tr> <tr> <td><code>boolean hasConnectionInformation()</code></td> <td>This returns true if connection information were provided. The UDF script can then obtain the connection information via <code>getConnectionInformation()</code>.</td> </tr> <tr> <td><code>ExaConnectionInformation getConnectionInformation()</code></td> <td>If <code>hasConnectionInformation()</code> is true, this returns the connection information provided by the user. See above in this table for the definition of <code>ExaConnectionInformation</code>.</td> </tr> </table> <p> The password is transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from <code>connection_name</code> field). The actual connection information can be obtained via <code>ExaMetadata.getConnection(connectionName)</code>.</p> <p> This function has to be implemented within the script main class.</p>	<code>Map<String, String> getParameters()</code>	Parameters specified in the IMPORT statement.	<code>boolean isSubselect()</code>	This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.	<code>List<String> getSubselectColumnNames()</code>	If <code>isSubselect()</code> is true and the user specified the target column names and types, this returns the names of all specified columns.	<code>List<String> getSubselectColumnTypes()</code>	If <code>isSubselect()</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").	<code>boolean hasConnectionName()</code>	This method returns true if a connection was specified. The UDF script can then obtain the connection information via <code>ExaMetadata.getConnection(name)</code> .	<code>String getConnectionName()</code>	If <code>hasConnection()</code> is true, this returns the name of the specified connection.	<code>boolean hasConnectionInformation()</code>	This returns true if connection information were provided. The UDF script can then obtain the connection information via <code>getConnectionInformation()</code> .	<code>ExaConnectionInformation getConnectionInformation()</code>	If <code>hasConnectionInformation()</code> is true, this returns the connection information provided by the user. See above in this table for the definition of <code>ExaConnectionInformation</code> .
<code>Map<String, String> getParameters()</code>	Parameters specified in the IMPORT statement.																
<code>boolean isSubselect()</code>	This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.																
<code>List<String> getSubselectColumnNames()</code>	If <code>isSubselect()</code> is true and the user specified the target column names and types, this returns the names of all specified columns.																
<code>List<String> getSubselectColumnTypes()</code>	If <code>isSubselect()</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").																
<code>boolean hasConnectionName()</code>	This method returns true if a connection was specified. The UDF script can then obtain the connection information via <code>ExaMetadata.getConnection(name)</code> .																
<code>String getConnectionName()</code>	If <code>hasConnection()</code> is true, this returns the name of the specified connection.																
<code>boolean hasConnectionInformation()</code>	This returns true if connection information were provided. The UDF script can then obtain the connection information via <code>getConnectionInformation()</code> .																
<code>ExaConnectionInformation getConnectionInformation()</code>	If <code>hasConnectionInformation()</code> is true, this returns the connection information provided by the user. See above in this table for the definition of <code>ExaConnectionInformation</code> .																

User defined export callback function <code>generateSqlForExportSpec(ExaMetadata meta, ExaExportSpecification export_spec)</code> (see Section 3.4.5, “User-defined EXPORT using UDFs”)	To support a user defined export you can implement the callback method <code>public static String generateSqlForExportSpec(ExaMetadata meta, ExaExportSpecification export_spec)</code> . Please see also Dynamic input and output parameters and the EXPORT statement for the syntax. The parameter <code>export_spec</code> contains all information about the executed EXPORT INTO SCRIPT statement. The function has to generate and return a SELECT SQL statement which will generate the data to be exported. The FROM part of that string can be a dummy table (e.g. DUAL) since the export command is aware which table should be exported. But it has to be specified to be able to compile the SQL string successfully.
	<code>export_spec</code> is an object with the following fields:
<code>Map<String, String> getParameters()</code>	Parameters specified in the EXPORT statement.
<code>List<String> getSourceColumnNames()</code>	List of column names of the resulting table that shall be exported.
<code>boolean hasTruncate()</code>	Boolean value from the EXPORT command option that defines whether the content of the target table should be truncated before the data transfer.
<code>boolean hasReplace()</code>	Boolean value from the EXPORT command option that defines whether the target table should be deleted before the data transfer.
<code>boolean hasCreatedBy()</code>	Boolean value from the EXPORT command option that defines whether a creation text was specified or not.
<code>String getCreatedBy()</code>	String value from the EXPORT command option that defines the creation text that is executed in the target system before the data transfer.
<code>boolean hasConnectionName()</code>	This method returns true if a connection was specified.
<code>String getConnectionName()</code>	If <code>hasConnection()</code> is true, this returns the name of the specified connection.
<code>boolean hasConnectionInformation()</code>	This returns true if connection information were provided. The UDF script can then obtain the connection information via <code>getConnectionInformation()</code> .
<code>ExaConnectionInformation getConnectionInformation()</code>	If <code>hasConnectionInformation()</code> is true, this returns the connection information provided by the user. See above in this table for the definition of <code>ExaConnectionInformation</code> .
<p> The password is transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from <code>connection_name</code> field). The actual connection information can be obtained via <code>ExaMetadata.getConnection(connectionName)</code>.</p>	
<p> This function has to be implemented within the script main class.</p>	

Example:

```
/*
This example loads from a webserver
and processes the following file goalies.xml:

<?xml version='1.0' encoding='UTF-8'?>
<users>
    <user active="1">
        <first_name>Manuel</first_name>
        <last_name>Neuer</last_name>
    </user>
```

```

<user active="1">
    <first_name>Joe</first_name>
    <last_name>Hart</last_name>
</user>
<user active="0">
    <first_name>Oliver</first_name>
    <last_name>Kahn</last_name>
</user>
</users>
*/
CREATE JAVA SCALAR SCRIPT process_users(url VARCHAR(500))
EMITS (firstname VARCHAR(20), lastname VARCHAR(20)) AS

import java.net.URL;
import java.netURLConnection;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;

class PROCESS_USERS {
    static void run(ExaMetadata exa, ExaIterator ctx) throws Exception {
        URL url = new URL(ctx.getString("url"));
        URLConnection conn = url.openConnection();
        DocumentBuilder docBuilder =
            DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document doc = docBuilder.parse(conn.getInputStream());
        NodeList nodes =
            doc.getDocumentElement().getElementsByTagName("user");
        for (int i = 0; i < nodes.getLength(); i++) {
            if (nodes.item(i).getNodeType() != Node.ELEMENT_NODE)
                continue;
            Element elem = (Element)nodes.item(i);
            if (!elem.getAttribute("active").equals("1"))
                continue;
            Node name = elem.getElementsByTagName("first_name").item(0);
            String firstName = name.getChildNodes().item(0).getNodeValue();
            name = elem.getElementsByTagName("last_name").item(0);
            String lastName = name.getChildNodes().item(0).getNodeValue();
            ctx.emit(firstName, lastName);
        }
    }
}
/
SELECT process_users ('http://www.my_valid_webserver/goalies.xml')
FROM DUAL;

FIRSTNAME          LASTNAME
-----  -----
Manuel             Neuer
Joe               Hart

```

Python

Beside the following information you can find additional details for Python in the official documentation (see <http://www.python.org>).

<code>run()</code> and <code>cleanup()</code> methods	<p>The method <code>run()</code> is called for each input tuple (SCALAR) or each group (SET). Its parameter is a kind of execution context and provides access to the data and the iterator in case of a SET script.</p> <p>To initialize expensive steps (e.g. opening external connections), you can write code outside the <code>run()</code> method, since this code is executed once at the beginning by each virtual machine. For deinitialization purpose, the method <code>cleanup()</code> exists which is called once for each virtual machine, at the end of the execution.</p>																																
Parameters	<p>Note that the internal Python data types and the database SQL types are not identical. Therefore casts must be done for the input and output data:</p> <table> <tbody> <tr><td><code>DECIMAL(p, 0)</code></td><td><code>int</code></td></tr> <tr><td><code>DECIMAL(p, s)</code></td><td><code>decimal.Decimal</code></td></tr> <tr><td><code>DOUBLE</code></td><td><code>float</code></td></tr> <tr><td><code>DATE</code></td><td><code>datetime.date</code></td></tr> <tr><td><code>TIMESTAMP</code></td><td><code>datetime.datetime</code></td></tr> <tr><td><code>BOOLEAN</code></td><td><code>bool</code></td></tr> <tr><td><code>VARCHAR</code> and <code>CHAR</code></td><td><code>unicode</code></td></tr> <tr><td>Other</td><td>Not supported</td></tr> </tbody> </table> <p> For performance reasons you should prefer <code>DOUBLE</code> to <code>DECIMAL</code> for the parameter types.</p> <p> The value <code>None</code> is the equivalent of the SQL <code>NULL</code>.</p> <p>The input parameters can be addressed by their names, e.g. <code>ctx.my_input</code>.</p> <p>But you can also use a dynamic number of parameters via the notation <code>(. . .)</code>, e.g. <code>CREATE PYTHON SCALAR SCRIPT my_script (. . .)</code>. The parameters can then be accessed through an index, e.g. <code>data[0]</code> for the first parameter. The number of parameters and their data types (both determined during the call of the script) are part of the metadata.</p>	<code>DECIMAL(p, 0)</code>	<code>int</code>	<code>DECIMAL(p, s)</code>	<code>decimal.Decimal</code>	<code>DOUBLE</code>	<code>float</code>	<code>DATE</code>	<code>datetime.date</code>	<code>TIMESTAMP</code>	<code>datetime.datetime</code>	<code>BOOLEAN</code>	<code>bool</code>	<code>VARCHAR</code> and <code>CHAR</code>	<code>unicode</code>	Other	Not supported																
<code>DECIMAL(p, 0)</code>	<code>int</code>																																
<code>DECIMAL(p, s)</code>	<code>decimal.Decimal</code>																																
<code>DOUBLE</code>	<code>float</code>																																
<code>DATE</code>	<code>datetime.date</code>																																
<code>TIMESTAMP</code>	<code>datetime.datetime</code>																																
<code>BOOLEAN</code>	<code>bool</code>																																
<code>VARCHAR</code> and <code>CHAR</code>	<code>unicode</code>																																
Other	Not supported																																
Metadata	<p>The following metadata can be accessed via global variables:</p> <table> <tbody> <tr><td><code>exa.meta.database_name</code></td><td>Database name</td></tr> <tr><td><code>exa.meta.database_version</code></td><td>Database version</td></tr> <tr><td><code>exa.meta.script_language</code></td><td>Name and version of the script language</td></tr> <tr><td><code>exa.meta.script_name</code></td><td>Name of the script</td></tr> <tr><td><code>exa.meta.script_schema</code></td><td>Schema in which the script is stored</td></tr> <tr><td><code>exa.meta.current_schema</code></td><td>Schema which is currently opened</td></tr> <tr><td><code>exa.meta.script_code</code></td><td>Code of the script</td></tr> <tr><td><code>exa.meta.session_id</code></td><td>Session ID</td></tr> <tr><td><code>exa.meta.statement_id</code></td><td>Statement ID within the session</td></tr> <tr><td><code>exa.meta.current_user</code></td><td>Current user</td></tr> <tr><td><code>exa.meta.node_count</code></td><td>Number of cluster nodes</td></tr> <tr><td><code>exa.meta.node_id</code></td><td>Local node ID starting with 0</td></tr> <tr><td><code>exa.meta.vm_id</code></td><td>Unique ID for the local machine (the IDs of the virtual machines have no relation to each other)</td></tr> <tr><td><code>exa.meta.input_type</code></td><td>Type of the input data (SCALAR or SET)</td></tr> <tr><td><code>exa.meta.input_column_count</code></td><td>Number of input columns</td></tr> <tr><td><code>exa.meta.input_columns[]</code></td><td>Array including the following information: {name, type, sql_type, precision, scale, length}</td></tr> </tbody> </table>	<code>exa.meta.database_name</code>	Database name	<code>exa.meta.database_version</code>	Database version	<code>exa.meta.script_language</code>	Name and version of the script language	<code>exa.meta.script_name</code>	Name of the script	<code>exa.meta.script_schema</code>	Schema in which the script is stored	<code>exa.meta.current_schema</code>	Schema which is currently opened	<code>exa.meta.script_code</code>	Code of the script	<code>exa.meta.session_id</code>	Session ID	<code>exa.meta.statement_id</code>	Statement ID within the session	<code>exa.meta.current_user</code>	Current user	<code>exa.meta.node_count</code>	Number of cluster nodes	<code>exa.meta.node_id</code>	Local node ID starting with 0	<code>exa.meta.vm_id</code>	Unique ID for the local machine (the IDs of the virtual machines have no relation to each other)	<code>exa.meta.input_type</code>	Type of the input data (SCALAR or SET)	<code>exa.meta.input_column_count</code>	Number of input columns	<code>exa.meta.input_columns[]</code>	Array including the following information: {name, type, sql_type, precision, scale, length}
<code>exa.meta.database_name</code>	Database name																																
<code>exa.meta.database_version</code>	Database version																																
<code>exa.meta.script_language</code>	Name and version of the script language																																
<code>exa.meta.script_name</code>	Name of the script																																
<code>exa.meta.script_schema</code>	Schema in which the script is stored																																
<code>exa.meta.current_schema</code>	Schema which is currently opened																																
<code>exa.meta.script_code</code>	Code of the script																																
<code>exa.meta.session_id</code>	Session ID																																
<code>exa.meta.statement_id</code>	Statement ID within the session																																
<code>exa.meta.current_user</code>	Current user																																
<code>exa.meta.node_count</code>	Number of cluster nodes																																
<code>exa.meta.node_id</code>	Local node ID starting with 0																																
<code>exa.meta.vm_id</code>	Unique ID for the local machine (the IDs of the virtual machines have no relation to each other)																																
<code>exa.meta.input_type</code>	Type of the input data (SCALAR or SET)																																
<code>exa.meta.input_column_count</code>	Number of input columns																																
<code>exa.meta.input_columns[]</code>	Array including the following information: {name, type, sql_type, precision, scale, length}																																

	<p><code>exa.meta.output_type</code> <code>exa.meta.output_column_count</code> <code>exa.meta.output_columns[]</code></p>	<p>Type of the output data (RETURNS or EMITS) Number of output columns Array including the following information: { name, type, sql_type, precision, scale, length }</p>																
<code>Data iterator, next(), size() and reset()</code>	<p>For scripts having multiple input tuples per call (keyword SET), you can iterate through that data via the method <code>next()</code>, which is accessible through the context. Initially, the iterator points to the first input row. For iterating you can use a <code>while True</code> loop which is aborted in case <code>if not ctx.next()</code>.</p> <p> If the input data is empty, then the <code>run()</code> method will not be called, and similar to aggregate functions the NULL value is returned as result (like for e.g. <code>SELECT MAX(x) FROM t WHERE false</code>).</p> <p>Additionally, there exist a method <code>reset()</code> which resets the iterator to the first input element. Hereby you can do multiple iterations through the data, if this is necessary for your algorithm.</p> <p>The method <code>size()</code> returns the number of input values.</p>																	
<code>emit()</code>	<p>You can return multiple output tuples per call (keyword EMITS) via the method <code>emit()</code>. The method expects as many parameters as output columns were defined. In the case of dynamic output parameters, it is handy in Python to use a <code>list</code> object which can be referenced using <code>*</code> (like in the example above: <code>ctx.emit(*currentRow)</code>).</p>																	
Import of other scripts	<p>Other scripts can be imported via the method <code>exa.import_script()</code>. The return value of this method must be assigned to a variable, representing the imported module.</p>																	
Accessing connection definitions	<p>The data that has been specified when defining connections with CREATE CONNECTION is available in Python UDF scripts via the method <code>exa.get_connection("<connection_name>")</code>. The result is a Python object with the following fields:</p> <table> <tr> <td><code>type</code></td><td>The type of the connection definition. ConnectionType is an enumeration which currently only contains the entry PASSWORD.</td></tr> <tr> <td><code>address</code></td><td>The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.</td></tr> <tr> <td><code>user</code></td><td>The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.</td></tr> <tr> <td><code>password</code></td><td>The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.</td></tr> </table>	<code>type</code>	The type of the connection definition. ConnectionType is an enumeration which currently only contains the entry PASSWORD.	<code>address</code>	The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.	<code>user</code>	The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.	<code>password</code>	The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.									
<code>type</code>	The type of the connection definition. ConnectionType is an enumeration which currently only contains the entry PASSWORD.																	
<code>address</code>	The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.																	
<code>user</code>	The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.																	
<code>password</code>	The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.																	
Auxiliary libraries	<p>The following libraries are provided which are not already part of the language:</p> <table> <tr> <td><code>cjson</code></td><td>Processing of JSON objects (see also http://pypi.python.org/pypi/python-cjson)</td></tr> <tr> <td><code>lxml</code></td><td>XML processing (see also http://pypi.python.org/pypi/lxml)</td></tr> <tr> <td><code>NumPy</code></td><td>Numeric calculations (see also http://www.scipy.org)</td></tr> <tr> <td><code>PyTables</code></td><td>Hierarchical database package (see also http://www.pytables.org). For this library, we added the necessary build package HDF5 (http://www.h5py.org).</td></tr> <tr> <td><code>pytz</code></td><td>Time zone functions (see also http://pytz.sourceforge.net)</td></tr> <tr> <td><code>redis</code></td><td>Interface for Redis (see also http://pypi.python.org/pypi/redis/)</td></tr> <tr> <td><code>scikit-learn</code></td><td>Machine Learning (see also http://scikit-learn.org)</td></tr> <tr> <td><code>SciPy</code></td><td>Scientific tools (see also http://www.scipy.org). For this library, we added the necessary build tool atlas (http://pypi.python.org/pypi/atlas)</td></tr> </table>	<code>cjson</code>	Processing of JSON objects (see also http://pypi.python.org/pypi/python-cjson)	<code>lxml</code>	XML processing (see also http://pypi.python.org/pypi/lxml)	<code>NumPy</code>	Numeric calculations (see also http://www.scipy.org)	<code>PyTables</code>	Hierarchical database package (see also http://www.pytables.org). For this library, we added the necessary build package HDF5 (http://www.h5py.org).	<code>pytz</code>	Time zone functions (see also http://pytz.sourceforge.net)	<code>redis</code>	Interface for Redis (see also http://pypi.python.org/pypi/redis/)	<code>scikit-learn</code>	Machine Learning (see also http://scikit-learn.org)	<code>SciPy</code>	Scientific tools (see also http://www.scipy.org). For this library, we added the necessary build tool atlas (http://pypi.python.org/pypi/atlas)	
<code>cjson</code>	Processing of JSON objects (see also http://pypi.python.org/pypi/python-cjson)																	
<code>lxml</code>	XML processing (see also http://pypi.python.org/pypi/lxml)																	
<code>NumPy</code>	Numeric calculations (see also http://www.scipy.org)																	
<code>PyTables</code>	Hierarchical database package (see also http://www.pytables.org). For this library, we added the necessary build package HDF5 (http://www.h5py.org).																	
<code>pytz</code>	Time zone functions (see also http://pytz.sourceforge.net)																	
<code>redis</code>	Interface for Redis (see also http://pypi.python.org/pypi/redis/)																	
<code>scikit-learn</code>	Machine Learning (see also http://scikit-learn.org)																	
<code>SciPy</code>	Scientific tools (see also http://www.scipy.org). For this library, we added the necessary build tool atlas (http://pypi.python.org/pypi/atlas)																	

Dynamic output parameters callback function <code>default_output_columns()</code>	<p>If the UDF script was defined with dynamic output parameters and the output parameters can not be determined (via specifying EMITS in the query or via INSERT INTO SELECT), the database calls the method <code>default_output_columns()</code> which you can implement. The expected return value is a String with the names and types of the output columns, e.g. "a int, b varchar(100)". See Dynamic input and output parameters for an explanation when this method is called including examples.</p> <p> You can access the Metadata <code>exa.meta</code> in the method, e.g. to find out the number and types of input columns.</p> <p> The method will be executed only once on a single node.</p>												
User defined import callback function <code>generate_sql_for_import_spec(import_spec)</code> (see Section 3.4.4, “User-defined IMPORT using UDFs”)	<p>To support a user defined import you can implement the callback method <code>generate_sql_for_import_spec(import_spec)</code>. Please see also Dynamic input and output parameters and the IMPORT statement for the syntax. The parameter <code>import_spec</code> contains all information about the executed IMPORT FROM SCRIPT statement. The function has to generate and return a SELECT SQL statement which will retrieve the data to be imported.</p> <p><code>import_spec</code> has the following fields:</p> <table> <tr> <td data-bbox="473 878 790 900"><code>parameters[]</code></td> <td data-bbox="890 878 1441 968">Parameters specified in the IMPORT statement. E.g. <code>parameters['FOO']</code> returns either the value of parameter FOO, or None, if FOO was not specified.</td> </tr> <tr> <td data-bbox="473 968 790 990"><code>is_subselect</code></td> <td data-bbox="890 968 1441 1057">This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.</td> </tr> <tr> <td data-bbox="473 1057 790 1080"><code>subselect_column_names[]</code></td> <td data-bbox="890 1057 1441 1147">If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the names of all specified columns.</td> </tr> <tr> <td data-bbox="473 1147 790 1170"><code>subselect_column_types[]</code></td> <td data-bbox="890 1147 1441 1282">If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").</td> </tr> <tr> <td data-bbox="473 1282 790 1304"><code>connection_name</code></td> <td data-bbox="890 1282 1441 1417">This returns the name of the connection, if such was specified. Otherwise it returns None. The UDF script can then obtain the connection information via <code>exa.get_connection(name)</code>.</td> </tr> <tr> <td data-bbox="473 1417 790 1439"><code>connection</code></td> <td data-bbox="890 1417 1441 1551">This is only defined, if the user provided connection information. It returns an object similar to such that is returned by <code>exa.getConnection(name)</code>. Returns None if no connection information are specified.</td> </tr> </table> <p> If a password was specified, it will be transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from the <code>connection_name</code> field). The actual connection information can be obtained via <code>exa.get_connection(name)</code>.</p>	<code>parameters[]</code>	Parameters specified in the IMPORT statement. E.g. <code>parameters['FOO']</code> returns either the value of parameter FOO, or None, if FOO was not specified.	<code>is_subselect</code>	This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.	<code>subselect_column_names[]</code>	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the names of all specified columns.	<code>subselect_column_types[]</code>	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").	<code>connection_name</code>	This returns the name of the connection, if such was specified. Otherwise it returns None. The UDF script can then obtain the connection information via <code>exa.get_connection(name)</code> .	<code>connection</code>	This is only defined, if the user provided connection information. It returns an object similar to such that is returned by <code>exa.getConnection(name)</code> . Returns None if no connection information are specified.
<code>parameters[]</code>	Parameters specified in the IMPORT statement. E.g. <code>parameters['FOO']</code> returns either the value of parameter FOO, or None, if FOO was not specified.												
<code>is_subselect</code>	This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.												
<code>subselect_column_names[]</code>	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the names of all specified columns.												
<code>subselect_column_types[]</code>	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").												
<code>connection_name</code>	This returns the name of the connection, if such was specified. Otherwise it returns None. The UDF script can then obtain the connection information via <code>exa.get_connection(name)</code> .												
<code>connection</code>	This is only defined, if the user provided connection information. It returns an object similar to such that is returned by <code>exa.getConnection(name)</code> . Returns None if no connection information are specified.												
User defined export callback function <code>generate_sql_for_export_spec(export_spec)</code> (see Section 3.4.5, “User-defined EXPORT using UDFs”)	<p>To support a user defined export you can implement the callback method <code>generate_sql_for_export_spec(export_spec)</code>. Please see also Dynamic input and output parameters and the EXPORT statement for the syntax. The parameter <code>export_spec</code> contains all information about the executed EXPORT INTO SCRIPT statement. The function has to generate and return a SELECT SQL statement which will generate the data to be exported. The FROM part of that string can be a dummy table (e.g. DUAL) since</p>												

defined EXPORT using UDFs															
<p>the export command is aware which table should be exported. But it has to be specified to be able to compile the SQL string successfully.</p> <p>export_spec has the following fields:</p> <table border="0"> <tr> <td>parameters[]</td><td>Parameters specified in the EXPORT statement. E.g. parameters['FOO'] returns either the value of parameter FOO, or None, if FOO was not specified.</td></tr> <tr> <td>source_column_names[]</td><td>List of column names of the resulting table that shall be exported.</td></tr> <tr> <td>has_truncate</td><td>Boolean value from the EXPORT command option that defines whether the content of the target table should be truncated before the data transfer.</td></tr> <tr> <td>has_replace</td><td>Boolean value from the EXPORT command option that defines whether the target table should be deleted before the data transfer.</td></tr> <tr> <td>created_by</td><td>String value from the EXPORT command option that defines the creation text that is executed in the target system before the data transfer.</td></tr> <tr> <td>connection_name</td><td>This returns the name of the connection, if such was specified. Otherwise it returns None. The UDF script can then obtain the connection information via exa.get_connection(name).</td></tr> <tr> <td>connection</td><td>This is only defined, if the user provided connection information. It returns an object similar to such that is returned by exa.get_connection(name). Returns None if no connection information are specified.</td></tr> </table> <p>! If a password was specified, it will be transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from the connection_name field). The actual connection information can be obtained via exa.get_connection(name).</p>	parameters[]	Parameters specified in the EXPORT statement. E.g. parameters['FOO'] returns either the value of parameter FOO, or None, if FOO was not specified.	source_column_names[]	List of column names of the resulting table that shall be exported.	has_truncate	Boolean value from the EXPORT command option that defines whether the content of the target table should be truncated before the data transfer.	has_replace	Boolean value from the EXPORT command option that defines whether the target table should be deleted before the data transfer.	created_by	String value from the EXPORT command option that defines the creation text that is executed in the target system before the data transfer.	connection_name	This returns the name of the connection, if such was specified. Otherwise it returns None. The UDF script can then obtain the connection information via exa.get_connection(name).	connection	This is only defined, if the user provided connection information. It returns an object similar to such that is returned by exa.get_connection(name). Returns None if no connection information are specified.	
parameters[]	Parameters specified in the EXPORT statement. E.g. parameters['FOO'] returns either the value of parameter FOO, or None, if FOO was not specified.														
source_column_names[]	List of column names of the resulting table that shall be exported.														
has_truncate	Boolean value from the EXPORT command option that defines whether the content of the target table should be truncated before the data transfer.														
has_replace	Boolean value from the EXPORT command option that defines whether the target table should be deleted before the data transfer.														
created_by	String value from the EXPORT command option that defines the creation text that is executed in the target system before the data transfer.														
connection_name	This returns the name of the connection, if such was specified. Otherwise it returns None. The UDF script can then obtain the connection information via exa.get_connection(name).														
connection	This is only defined, if the user provided connection information. It returns an object similar to such that is returned by exa.get_connection(name). Returns None if no connection information are specified.														

Example:

```
/*
This example loads from a webserver
and processes the following file goalies.xml:

<?xml version='1.0' encoding='UTF-8'?>
<users>
    <user active="1">
        <first_name>Manuel</first_name>
        <last_name>Neuer</last_name>
    </user>
    <user active="1">
        <first_name>Joe</first_name>
        <last_name>Hart</last_name>
    </user>
    <user active="0">
        <first_name>Oliver</first_name>
        <last_name>Kahn</last_name>
    </user>
</users>
```

```
/*
CREATE PYTHON SCALAR SCRIPT process_users(url VARCHAR(500))
EMITS (firstname VARCHAR(20), lastname VARCHAR(20)) AS
import re
import xml.etree.cElementTree as etree
from urllib2 import urlopen

def run(ctx):
    content = etree.parse(urlopen(ctx.url))
    for user in content.findall('user[@active="1"]'):
        ctx.emit(user.find('first_name').text, user.find('last_name').text)
/

SELECT process_users ('http://www.my_valid_webserver/goalies.xml')
FROM DUAL;

FIRSTNAME          LASTNAME
-----
Manuel             Neuer
Joe                Hart
```

R

Beside the following information you can find additional details for R in the official documentation (see <http://www.r-project.org>).

<code>run()</code> and <code>cleanup()</code> methods	<p>The method <code>run()</code> is called for each input tuple (SCALAR) or each group (SET). Its parameter is a kind of execution context and provides access to the data and the iterator in case of a SET script.</p> <p>To initialize expensive steps (e.g. opening external connections), you can write code outside the <code>run()</code> method, since this code is executed once at the beginning by each virtual machine. For deinitialization purpose, the method <code>cleanup()</code> exists which is called once for each virtual machine, at the end of the execution.</p>												
Parameters	<p>Note that the internal R data types and the database SQL types are not identical. Therefore casts must be done for the input and output data:</p> <table> <tbody> <tr> <td><code>DECIMAL(p, 0)</code> for $p \leq 9$</td> <td><code>integer</code></td> </tr> <tr> <td><code>DECIMAL(p, s)</code> and <code>DOUBLE</code></td> <td><code>double</code></td> </tr> <tr> <td><code>DATE</code> and <code>TIMESTAMP</code></td> <td><code>POSIXt</code></td> </tr> <tr> <td><code>BOOLEAN</code></td> <td><code>logical</code></td> </tr> <tr> <td><code>VARCHAR</code> and <code>CHAR</code></td> <td><code>character</code></td> </tr> <tr> <td>Other</td> <td>Not supported</td> </tr> </tbody> </table> <p> For performance reasons you should prefer <code>DOUBLE</code> to <code>DECIMAL(p,s)</code> for the parameter types.</p> <p> The value <code>NA</code> is the equivalent of the SQL <code>NULL</code>.</p> <p>The input parameters can be addressed by their names, e.g. <code>ctx\$my_input</code>.</p> <p>But you can also use a dynamic number of parameters via the notation <code>(. . .)</code>, e.g. <code>CREATE R SCALAR SCRIPT my_script (. . .)</code>. The parameters can then be accessed through an index. Please note that in this case a special notation is necessary, e.g. <code>ctx[[1]]()</code></p>	<code>DECIMAL(p, 0)</code> for $p \leq 9$	<code>integer</code>	<code>DECIMAL(p, s)</code> and <code>DOUBLE</code>	<code>double</code>	<code>DATE</code> and <code>TIMESTAMP</code>	<code>POSIXt</code>	<code>BOOLEAN</code>	<code>logical</code>	<code>VARCHAR</code> and <code>CHAR</code>	<code>character</code>	Other	Not supported
<code>DECIMAL(p, 0)</code> for $p \leq 9$	<code>integer</code>												
<code>DECIMAL(p, s)</code> and <code>DOUBLE</code>	<code>double</code>												
<code>DATE</code> and <code>TIMESTAMP</code>	<code>POSIXt</code>												
<code>BOOLEAN</code>	<code>logical</code>												
<code>VARCHAR</code> and <code>CHAR</code>	<code>character</code>												
Other	Not supported												

	for the first parameter. The number of parameters and their data types (both determined during the call of the script) are part of the metadata.																																						
Metadata	<p>The following metadata can be accessed via global variables:</p> <table> <tbody> <tr><td>exa\$meta\$database_name</td><td>Database name</td></tr> <tr><td>exa\$meta\$database_version</td><td>Database version</td></tr> <tr><td>exa\$meta\$script_language</td><td>Name and version of the script language</td></tr> <tr><td>exa\$meta\$script_name</td><td>Name of the script</td></tr> <tr><td>exa\$meta\$script_schema</td><td>Schema in which the script is stored</td></tr> <tr><td>exa\$meta\$current_schema</td><td>Schema which is currently opened</td></tr> <tr><td>exa\$meta\$script_code</td><td>Code of the script</td></tr> <tr><td>exa\$meta\$session_id</td><td>Session id</td></tr> <tr><td>exa\$meta\$statement_id</td><td>Statement ID within the session</td></tr> <tr><td>exa\$meta\$current_user</td><td>Current user</td></tr> <tr><td>exa\$meta\$node_count</td><td>Number of cluster nodes</td></tr> <tr><td>exa\$meta\$node_id</td><td>Local node ID starting with 0</td></tr> <tr><td>exa\$meta\$vm_id</td><td>Unique ID for the local machine (the IDs of the virtual machines have no relation to each other)</td></tr> <tr><td>exa\$meta\$input_type</td><td>Type of the input data (SCALAR or SET)</td></tr> <tr><td>exa\$meta\$input_column_count</td><td>Number of input columns</td></tr> <tr><td>exa\$meta\$input_columns[]</td><td>Array including the following information: {name, type, sql_type, precision, scale, length}</td></tr> <tr><td>exa\$meta\$output_type</td><td>Type of the output data (RETURNS or EMITS)</td></tr> <tr><td>exa\$meta\$output_column_count</td><td>Number of output columns</td></tr> <tr><td>exa\$meta\$output_columns[]</td><td>Array including the following information: {name, type, sql_type, precision, scale, length}</td></tr> </tbody> </table>	exa\$meta\$database_name	Database name	exa\$meta\$database_version	Database version	exa\$meta\$script_language	Name and version of the script language	exa\$meta\$script_name	Name of the script	exa\$meta\$script_schema	Schema in which the script is stored	exa\$meta\$current_schema	Schema which is currently opened	exa\$meta\$script_code	Code of the script	exa\$meta\$session_id	Session id	exa\$meta\$statement_id	Statement ID within the session	exa\$meta\$current_user	Current user	exa\$meta\$node_count	Number of cluster nodes	exa\$meta\$node_id	Local node ID starting with 0	exa\$meta\$vm_id	Unique ID for the local machine (the IDs of the virtual machines have no relation to each other)	exa\$meta\$input_type	Type of the input data (SCALAR or SET)	exa\$meta\$input_column_count	Number of input columns	exa\$meta\$input_columns[]	Array including the following information: {name, type, sql_type, precision, scale, length}	exa\$meta\$output_type	Type of the output data (RETURNS or EMITS)	exa\$meta\$output_column_count	Number of output columns	exa\$meta\$output_columns[]	Array including the following information: {name, type, sql_type, precision, scale, length}
exa\$meta\$database_name	Database name																																						
exa\$meta\$database_version	Database version																																						
exa\$meta\$script_language	Name and version of the script language																																						
exa\$meta\$script_name	Name of the script																																						
exa\$meta\$script_schema	Schema in which the script is stored																																						
exa\$meta\$current_schema	Schema which is currently opened																																						
exa\$meta\$script_code	Code of the script																																						
exa\$meta\$session_id	Session id																																						
exa\$meta\$statement_id	Statement ID within the session																																						
exa\$meta\$current_user	Current user																																						
exa\$meta\$node_count	Number of cluster nodes																																						
exa\$meta\$node_id	Local node ID starting with 0																																						
exa\$meta\$vm_id	Unique ID for the local machine (the IDs of the virtual machines have no relation to each other)																																						
exa\$meta\$input_type	Type of the input data (SCALAR or SET)																																						
exa\$meta\$input_column_count	Number of input columns																																						
exa\$meta\$input_columns[]	Array including the following information: {name, type, sql_type, precision, scale, length}																																						
exa\$meta\$output_type	Type of the output data (RETURNS or EMITS)																																						
exa\$meta\$output_column_count	Number of output columns																																						
exa\$meta\$output_columns[]	Array including the following information: {name, type, sql_type, precision, scale, length}																																						
Data iterator, <code>next_row()</code> , <code>size()</code> and <code>reset()</code>	<p>For scripts having multiple input tuples per call (keyword SET) there are two ways to access the data of your group:</p> <ol style="list-style-type: none"> 1. Iterate over the data, one record at a time 2. Load all (or multiple) rows for the group into a vector in memory to run R operations on it <p>To iterate over the data, use the method <code>next_row()</code> which is accessible through the context. The function differs from the other languages, because <code>next</code> is a reserved keyword in R.</p> <p>Initially, the iterator points to the first input row. That's why a <i>repeat...until</i> loop can be ideally used to iterate through the data (see examples). If the input data is empty, then the <code>run()</code> method will not be called, and similar to aggregate functions the NULL value is returned as result (like for e.g. <code>SELECT MAX(x) FROM t WHERE false</code>).</p> <p>To access the data as a vector, you can call the method <code>next_row</code> with a parameter that specifies how many records you want to read into the vector (e.g. <code>next_row(1000)</code> or <code>next_row(NA)</code>). If you specify NA, all records of the group will be read into a single vector.</p> <p>Please keep in mind that the vector will be held completely in memory. For very large groups this might exceed the memory limits, and you should fetch e.g. 1000 rows at a time until you processed all records of the group.</p> <p>To get the actual vector with data, you can access the context property with the name of your input parameter (e.g. <code>ctx\$input_word</code>, see the example below). Note that you have to call the <code>next_row</code> function first in this case, before accessing the data.</p>																																						

	<p>Additionally, there exist a method <code>reset()</code> which resets the iterator to the first input element. Hereby you can do multiple iterations through the data, if this is necessary for your algorithm.</p> <p>The method <code>size()</code> returns the number of input values.</p>																								
<code>emit()</code>	<p>You can return multiple output tuples per call (keyword EMITS) via the method <code>emit()</code>.</p> <p>The method expects as many parameters as output columns were defined. In the case of dynamic output parameters, it is handy to use a list and the <code>do.call</code> method like in the following example: <code>do.call(ctx\$emit, list(1, "a"))</code></p> <p>For scripts of type RETURNS, vectors can be used to improve the performance (see examples below).</p>																								
Import of other scripts	Other scripts can be imported via the method <code>exa\$import_script()</code> . The return value of this method must be assigned to a variable, representing the imported module.																								
Accessing connection definitions	<p>The data that has been specified when defining connections with CREATE CONNECTION is available in R UDF scripts via the method <code>exa\$get_connection("connection_name")</code>. The result is an R list with the following entries:</p> <table> <tr> <td>type</td><td>The type of the connection definition. ConnectionType is an enumeration which currently only contains the entry PASSWORD.</td></tr> <tr> <td>address</td><td>The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.</td></tr> <tr> <td>user</td><td>The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.</td></tr> <tr> <td>password</td><td>The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.</td></tr> </table>	type	The type of the connection definition. ConnectionType is an enumeration which currently only contains the entry PASSWORD.	address	The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.	user	The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.	password	The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.																
type	The type of the connection definition. ConnectionType is an enumeration which currently only contains the entry PASSWORD.																								
address	The part of the connection definition that followed the TO keyword in the CREATE CONNECTION command.																								
user	The part of the connection definition that followed the USER keyword in the CREATE CONNECTION command.																								
password	The part of the connection definition that followed the IDENTIFIED BY keyword in the CREATE CONNECTION command.																								
Auxiliary libraries	<p>The following libraries are provided which are not already part of the language:</p> <table> <tr> <td>bitops</td><td>Bitwise operations on integer vectors (see also http://cran.r-project.org/web/packages/bitops)</td></tr> <tr> <td>class</td><td>Various functions for classification (see also http://cran.r-project.org/web/packages/class)</td></tr> <tr> <td>data.table</td><td>Extension of <code>data.frame</code> for fast indexing, fast ordered joins, fast assignment, fast grouping and list columns (see also http://cran.r-project.org/web/packages/data.table)</td></tr> <tr> <td>e1071</td><td>Misc functions of the Department of Statistics (e1071) of the university TU Wien (see also http://cran.r-project.org/web/packages/e1071)</td></tr> <tr> <td>flashClust</td><td>Implementation of optimal hierarchical clustering (see also http://cran.r-project.org/web/packages/flashClust)</td></tr> <tr> <td>gbm</td><td>Generalized Boosted Regression Models (see also http://cran.r-project.org/web/packages/gbm)</td></tr> <tr> <td>Hmisc</td><td>Harrell Miscellaneous (see also http://cran.r-project.org/web/packages/Hmisc)</td></tr> <tr> <td>MASS</td><td>Functions and data sets for Venables and Ripley's MASS (see also http://cran.r-project.org/web/packages/MASS)</td></tr> <tr> <td>randomForest</td><td>Classification and regression algorithms (see also http://cran.r-project.org/web/packages/randomForest)</td></tr> <tr> <td>RCurl</td><td>Support for internet access like http connections (see also http://www.omegahat.org/RCurl)</td></tr> <tr> <td>RODBC</td><td>Interface for ODBC databases (see also http://cran.r-project.org/web/packages/RODBC)</td></tr> <tr> <td>rpart</td><td>Recursive partitioning and regression trees (see also http://cran.r-project.org/web/packages/rpart)</td></tr> </table>	bitops	Bitwise operations on integer vectors (see also http://cran.r-project.org/web/packages/bitops)	class	Various functions for classification (see also http://cran.r-project.org/web/packages/class)	data.table	Extension of <code>data.frame</code> for fast indexing, fast ordered joins, fast assignment, fast grouping and list columns (see also http://cran.r-project.org/web/packages/data.table)	e1071	Misc functions of the Department of Statistics (e1071) of the university TU Wien (see also http://cran.r-project.org/web/packages/e1071)	flashClust	Implementation of optimal hierarchical clustering (see also http://cran.r-project.org/web/packages/flashClust)	gbm	Generalized Boosted Regression Models (see also http://cran.r-project.org/web/packages/gbm)	Hmisc	Harrell Miscellaneous (see also http://cran.r-project.org/web/packages/Hmisc)	MASS	Functions and data sets for Venables and Ripley's MASS (see also http://cran.r-project.org/web/packages/MASS)	randomForest	Classification and regression algorithms (see also http://cran.r-project.org/web/packages/randomForest)	RCurl	Support for internet access like http connections (see also http://www.omegahat.org/RCurl)	RODBC	Interface for ODBC databases (see also http://cran.r-project.org/web/packages/RODBC)	rpart	Recursive partitioning and regression trees (see also http://cran.r-project.org/web/packages/rpart)
bitops	Bitwise operations on integer vectors (see also http://cran.r-project.org/web/packages/bitops)																								
class	Various functions for classification (see also http://cran.r-project.org/web/packages/class)																								
data.table	Extension of <code>data.frame</code> for fast indexing, fast ordered joins, fast assignment, fast grouping and list columns (see also http://cran.r-project.org/web/packages/data.table)																								
e1071	Misc functions of the Department of Statistics (e1071) of the university TU Wien (see also http://cran.r-project.org/web/packages/e1071)																								
flashClust	Implementation of optimal hierarchical clustering (see also http://cran.r-project.org/web/packages/flashClust)																								
gbm	Generalized Boosted Regression Models (see also http://cran.r-project.org/web/packages/gbm)																								
Hmisc	Harrell Miscellaneous (see also http://cran.r-project.org/web/packages/Hmisc)																								
MASS	Functions and data sets for Venables and Ripley's MASS (see also http://cran.r-project.org/web/packages/MASS)																								
randomForest	Classification and regression algorithms (see also http://cran.r-project.org/web/packages/randomForest)																								
RCurl	Support for internet access like http connections (see also http://www.omegahat.org/RCurl)																								
RODBC	Interface for ODBC databases (see also http://cran.r-project.org/web/packages/RODBC)																								
rpart	Recursive partitioning and regression trees (see also http://cran.r-project.org/web/packages/rpart)																								

	rredis RSXML survival	Interface for Redis (see also http://cran.r-project.org/web/packages/rredis) XML parsing (see also http://www.omegahat.org/RSXML) Survival Analysis (see also http://cran.r-project.org/web/packages/survival)											
Dynamic output parameters callback function <code>defaultOutputColumns()</code>	If the UDF script was defined with dynamic output parameters and the output parameters can not be determined (via specifying EMITS in the query or via INSERT INTO SELECT), the database calls the method <code>defaultOutputColumns()</code> which you can implement. The expected return value is a String with the names and types of the output columns, e.g. "a int, b varchar(100)". See Dynamic input and output parameters for an explanation when this method is called including examples.	<p> You can access the Metadata <code>exa\$meta</code> in the method, e.g. to find out the number and types of input columns.</p> <p> The method will be executed only once on a single node.</p>											
User defined import callback function <code>generate_sql_for_import_spec(import_spec)</code> (see Section 3.4.4, “User-defined IMPORT using UDFs”)	<p>To support a user defined import you can implement the callback method <code>generate_sql_for_import_spec(import_spec)</code>. Please see also Dynamic input and output parameters and the IMPORT statement for the syntax. The parameter <code>import_spec</code> contains all information about the executed IMPORT FROM SCRIPT statement. The function has to generate and return a SELECT SQL statement which will retrieve the data to be imported.</p> <p><code>import_spec</code> has the following fields:</p> <table> <tr> <td>parameters</td> <td>Parameters specified in the IMPORT statement. E.g. <code>parameters\$FOO</code> returns the value of parameter FOO.</td> </tr> <tr> <td>is_subselect</td> <td>This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.</td> </tr> <tr> <td>subselect_column_names[]</td> <td>If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the names of all specified columns.</td> </tr> <tr> <td>subselect_column_types[]</td> <td>If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").</td> </tr> <tr> <td>connection_name</td> <td>The name of the connection, if such was specified. The UDF script can then obtain the connection information via <code>exa\$get_connection(name)</code>. Otherwise it returns NULL.</td> </tr> <tr> <td>connection</td> <td>If the user provided connection information this returns an object with connection information similar to such that is returned by <code>exa.getConnection(name)</code>. Otherwise it returns NULL.</td> </tr> </table> <p> If a password was specified, it will be transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from the <code>connection_name</code> field). The actual connection information can be obtained via <code>exa\$get_connection(name)</code>.</p>	parameters	Parameters specified in the IMPORT statement. E.g. <code>parameters\$FOO</code> returns the value of parameter FOO.	is_subselect	This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.	subselect_column_names[]	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the names of all specified columns.	subselect_column_types[]	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").	connection_name	The name of the connection, if such was specified. The UDF script can then obtain the connection information via <code>exa\$get_connection(name)</code> . Otherwise it returns NULL.	connection	If the user provided connection information this returns an object with connection information similar to such that is returned by <code>exa.getConnection(name)</code> . Otherwise it returns NULL.
parameters	Parameters specified in the IMPORT statement. E.g. <code>parameters\$FOO</code> returns the value of parameter FOO.												
is_subselect	This is true, if the IMPORT is used inside a SELECT statement and not inside an IMPORT INTO table statement.												
subselect_column_names[]	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the names of all specified columns.												
subselect_column_types[]	If <code>is_subselect</code> is true and the user specified the target column names and types, this returns the types of all specified columns. The types are returned in SQL format (e.g. "VARCHAR(100)").												
connection_name	The name of the connection, if such was specified. The UDF script can then obtain the connection information via <code>exa\$get_connection(name)</code> . Otherwise it returns NULL.												
connection	If the user provided connection information this returns an object with connection information similar to such that is returned by <code>exa.getConnection(name)</code> . Otherwise it returns NULL.												

User defined export callback function <code>generate_sql_for_export_spec(export_spec)</code> (see Section 3.4.5, “User-defined EXPORT using UDFs”)	To support a user defined export you can implement the callback method <code>generate_sql_for_export_spec(export_spec)</code> . Please see also Dynamic input and output parameters and the <code>EXPORT</code> statement for the syntax. The parameter <code>export_spec</code> contains all information about the executed <code>EXPORT INTO SCRIPT</code> statement. The function has to generate and return a <code>SELECT</code> SQL statement which will generate the data to be exported. The <code>FROM</code> part of that string can be a dummy table (e.g. <code>DUAL</code>) since the export command is aware which table should be exported. But it has to be specified to be able to compile the SQL string successfully. <code>export_spec</code> has the following fields: <table> <tbody> <tr> <td><code>parameters</code></td><td>Parameters specified in the <code>IMPORT</code> statement. E.g. <code>parameters\$FOO</code> returns the value of parameter <code>FOO</code>.</td></tr> <tr> <td><code>source_column_names</code></td><td>List of column names of the resulting table that shall be exported.</td></tr> <tr> <td><code>has_truncate</code></td><td>Boolean value from the <code>EXPORT</code> command option that defines whether the content of the target table should be truncated before the data transfer.</td></tr> <tr> <td><code>has_replace</code></td><td>Boolean value from the <code>EXPORT</code> command option that defines whether the target table should be deleted before the data transfer.</td></tr> <tr> <td><code>created_by</code></td><td>String value from the <code>EXPORT</code> command option that defines the creation text that is executed in the target system before the data transfer.</td></tr> <tr> <td><code>connection_name</code></td><td>The name of the connection, if such was specified. The UDF script can then obtain the connection information via <code>exa\$get_connection(name)</code>. Otherwise it returns <code>NULL</code>.</td></tr> <tr> <td><code>connection</code></td><td>If the user provided connection information this returns an object with connection information similar to such that is returned by <code>exa.getConnection(name)</code>.Otherwise it returns <code>NULL</code>.</td></tr> </tbody> </table> <p> If a password was specified, it will be transferred clear-text and might be shown in the logs. We recommend to create a CONNECTION before and to specify only the connection name (can be obtained from the <code>connection_name</code> field). The actual connection information can be obtained via <code>exa\$get_connection(name)</code>.</p>	<code>parameters</code>	Parameters specified in the <code>IMPORT</code> statement. E.g. <code>parameters\$FOO</code> returns the value of parameter <code>FOO</code> .	<code>source_column_names</code>	List of column names of the resulting table that shall be exported.	<code>has_truncate</code>	Boolean value from the <code>EXPORT</code> command option that defines whether the content of the target table should be truncated before the data transfer.	<code>has_replace</code>	Boolean value from the <code>EXPORT</code> command option that defines whether the target table should be deleted before the data transfer.	<code>created_by</code>	String value from the <code>EXPORT</code> command option that defines the creation text that is executed in the target system before the data transfer.	<code>connection_name</code>	The name of the connection, if such was specified. The UDF script can then obtain the connection information via <code>exa\$get_connection(name)</code> . Otherwise it returns <code>NULL</code> .	<code>connection</code>	If the user provided connection information this returns an object with connection information similar to such that is returned by <code>exa.getConnection(name)</code> .Otherwise it returns <code>NULL</code> .
<code>parameters</code>	Parameters specified in the <code>IMPORT</code> statement. E.g. <code>parameters\$FOO</code> returns the value of parameter <code>FOO</code> .														
<code>source_column_names</code>	List of column names of the resulting table that shall be exported.														
<code>has_truncate</code>	Boolean value from the <code>EXPORT</code> command option that defines whether the content of the target table should be truncated before the data transfer.														
<code>has_replace</code>	Boolean value from the <code>EXPORT</code> command option that defines whether the target table should be deleted before the data transfer.														
<code>created_by</code>	String value from the <code>EXPORT</code> command option that defines the creation text that is executed in the target system before the data transfer.														
<code>connection_name</code>	The name of the connection, if such was specified. The UDF script can then obtain the connection information via <code>exa\$get_connection(name)</code> . Otherwise it returns <code>NULL</code> .														
<code>connection</code>	If the user provided connection information this returns an object with connection information similar to such that is returned by <code>exa.getConnection(name)</code> .Otherwise it returns <code>NULL</code> .														

Example:

```
-- A simple SCALAR RETURNS function, similar to the SQL function UPPER()
CREATE R SCALAR SCRIPT r_upper(input_word VARCHAR(50))
RETURNS VARCHAR(50) AS
run <- function(ctx) {
    # the last statement in a function is automatically used as return value
    toupper(ctx$input_word)
}
/

SELECT last_name, r_upper(last_name) AS r_upper FROM customer;

LAST_NAME R_UPPER
-----
Smith      SMITH
```

```

Miller      MILLER

-- The same function as r_upper, but using a faster vector operation
CREATE R SCALAR SCRIPT r_upper_vectorized(input_word VARCHAR(50))
RETURNS VARCHAR(50) AS
run <- function(ctx) {
    # read all records in the current buffer into a single vector.
    # the buffer holds any number of records from the table
    ctx$next_row(NA) toupper(ctx$input_word)
}
/

-- A simple SET-EMITS function, computing simple statistics on a column
CREATE R SET SCRIPT r_stats(group_id INT, input_number DOUBLE)
EMITS (group_id INT, mean DOUBLE, stddev DOUBLE) AS
run <- function(ctx) {
    # fetch all records from this group into a single vector
    ctx$next_row(NA)
    ctx$emit(ctx$group_id[1], mean(ctx$input_number), sd(ctx$input_number))
}
/

SELECT r_stats(groupId, num) FROM keyvalues
GROUP BY groupId ORDER BY group_id;

GROUP_ID MEAN          STDDEV
----- -----
1        1.0333333333 0.152752523
2        42              12.16552506059644

/*
The following example loads from a webserver
and processes the following file goalies.xml:

<?xml version='1.0' encoding='UTF-8'?>
<users>
    <user active="1">
        <first_name>Manuel</first_name>
        <last_name>Neuer</last_name>
    </user>
    <user active="1">
        <first_name>Joe</first_name>
        <last_name>Hart</last_name>
    </user>
    <user active="0">
        <first_name>Oliver</first_name>
        <last_name>Kahn</last_name>
    </user>
</users>
*/
CREATE R SCALAR SCRIPT process_users(url VARCHAR(500))
EMITS (firstname VARCHAR(100), lastname VARCHAR(100)) AS
require('RCurl')
require('XML')

run <- function(ctx) {

```

```

cont <- getURL(ctx$url)
tree <- xmlTreeParse(cont)
for (i in 1:length(tree$doc$children$users)) {
  if (tree$doc$children$users[i]$user$attributes['active']==1) {
    firstname <- tree$doc$children$users[i]$user$children$first_name
    lastname  <- tree$doc$children$users[i]$user$children$last_name
    ctx$emit(firstname$children$text$value,
             lastname$children$text$value)
  }
}
/
SELECT process_users ('http://www.my_valid_webserver/goalies.xml');

FIRSTNAME          LASTNAME
-----
Manuel            Neuer
Joe               Hart

```

3.6.4. The synchronous cluster file system BucketFS

When scripts are executed in parallel on the Exasol cluster, there exist some use cases where all instances have to access the same external data. Your algorithms could for example use a statistical model or weather data. For such requirements, it's obviously possible to use an external service (e.g. a file server). But in terms of performance, it would be quite handy to have such data available locally on the cluster nodes.

The Exasol BucketFS file system has been developed for such use cases, where data should be stored synchronously and replicated across the cluster. But we will also explain in the following sections how this concept can be used to extend script languages and even to install completely new script languages on the Exasol cluster.

If you are interested in a concrete practical data science use case, we recommend the following entry in our Solution Center: <https://wwwexasol.com/support/browse/SOL-257>

What is BucketFS?

The BucketFS file system is a synchronous file system which is available in the Exasol cluster. This means that each cluster node can connect to this service (e.g. through the http interface) and will see exactly the same content.



The data is replicated locally on every server and automatically synchronized. Hence, you shouldn't store large amounts of data there.



The data in BucketFS is not part of the database backups and has to be backed up manually if necessary.

One BucketFS service contains any number of so-called buckets, and every bucket stores any number of files. Every bucket can have different access privileges as we will explain later on. Folders are not supported directly, but if you specify an upload path including folders, these will be created transparently if they do not exist yet. If all files from a folder are deleted, the folder will be dropped automatically.

Writing data is an atomic operation. There don't exist any locks on files, so the latest write operation will finally overwrite the file. In contrast to the database itself, BucketFS is a pure file-based system and has no transactional semantic.

Setting up BucketFS and creating buckets

On the left part of the EXAoperation administration interface, you'll find the link to the BucketFS configuration. You will find a pre-installed default BucketFS service for the configured data disk. If you want to create additional file system services, you need to specify only the data disk and specify ports for http(s).

If you follow the link of an BucketFS Id, you can create and configure any number of buckets within this BucketFS. Beside the bucket name, you have to specify read/write passwords and define whether the bucket should be public readable, i.e. accessible for everyone.



A default bucket already exists in the default BucketFS which contains the pre-installed script languages (Java, Python, R). However, for storing larger user data we highly recommend to create a separate BucketFS instance on a separate partition.

Access and access control

From outside the cluster, it is possible to access the buckets and the contained files through http(s) clients such as curl™. You only have to use one of the database servers' IP address, the specified port and the bucket name, and potentially adjust your internal firewall configuration.



For accessing a bucket through http(s) the users `r` and `w` are always configured and are associated with the configured read and write passwords.

In the following example the http client curl is used to list the existing buckets, upload the files `file1` and `tar1.tgz` into the bucket `bucket1` and finally display the list of contained files in this bucket. The relevant parameters for our example are the port of the BucketFS (1234), the name of the bucket (`bucket1`) and the passwords (`readpw` and `writepw`).

```
$> curl http://192.168.6.75:1234
default
bucket1
$> curl -X PUT -T file1 http://w:writepw@192.168.6.75:1234/bucket1/file1
$> curl -X PUT -T tar1.tgz \
http://w:writepw@192.168.6.75:1234/bucket1/tar1.tgz
$> curl http://r:readpw@192.168.6.75:1234/bucket1
file1
tar1.tgz
```

From scripts running on the Exasol cluster, you can access the files locally for simplification reasons. You don't need to define any IP address and can be sure that the data is used from the local node. The corresponding path for a bucket can be found in EXAoperation in the overview of a bucket.

The access control is organized by using a database CONNECTION object (see also [CREATE CONNECTION](#)), because for the database, buckets look similar to normal external data sources. The connection contains the path to the bucket and the read password. After granting that connection to someone using the [GRANT](#) command, the bucket becomes visible/accessible for that user. If you want to allow all users to access a bucket, you can define that bucket in EXAoperation as *public*.



Similar to external clients, write access from scripts is only possible via http(s), but you still would have to be careful with the parallelism of script processes.

In the following example, a connection to a bucket is defined and granted. Afterwards, a script is created which lists the files from a local path. You can see in the example that the equivalent local path for the previously created bucket `bucket1` is `/buckets/bfsdefault/bucket1`.

```
CREATE CONNECTION my_bucket_access TO 'bucketfs:bfsdefault/bucket1'
IDENTIFIED BY 'readpw';
```

```
GRANT CONNECTION my_bucket_access TO my_user;

CREATE PYTHON SCALAR SCRIPT ls(my_path VARCHAR(100))
EMITS (files VARCHAR(100)) AS
import subprocess

def run(c):
    try:
        p = subprocess.Popen('ls '+c.my_path,
                            stdout      = subprocess.PIPE,
                            stderr      = subprocess.STDOUT,
                            close_fds   = True,
                            shell       = True)
        out, err = p.communicate()
        for line in out.strip().split('\n'):
            c.emit(line)
    finally:
        if p is not None:
            try: p.kill()
            except: pass
    /
SELECT ls('/buckets/bfsdefault/bucket1');

FILES
-----
file1
tar1

SELECT ls('/buckets/bfsdefault/bucket1/tar1/');

FILES
-----
a
b
```

As you might have recognized in the example, archives (.zip, .tar, .tar.gz or .tgz) are always extracted for the script access on the local file system. From outside (e.g. via curl) you see the archive while you can locally use the extracted files from within the scripts.



If you store archives (.zip, .tar, .tar.gz or .tgz) in the BucketFS, both the original files and the extracted files are stored and need therefore storage space twice.



If you want to work on an archive directly, you can avoid the extraction by renaming the file extension (e.g. .zipx instead of .zip).

3.6.5. Expanding script languages using BucketFS

If Exasol's preconfigured set of script languages is sufficient for your needs, you don't need to consider this chapter. But if you want to expand the script languages (e.g. installing additional R packages) or even integrate completely new languages into the script framework, you can easily do that using BucketFS.

Expanding the existing script languages

The first option is to expand the existing languages by adding further packages. The corresponding procedure differs for every language and will therefore be explained in the following sections.



The script language Lua is not expandable, because it is natively compiled into the Exasol database software.

Java files (.jar)

For Java, you can integrate .jar files in Exasol easily. You only have to save the file in a bucket and reference the corresponding path directly in your Java script.

If you for instance want to use Google's library to process telephone numbers (<http://mavensearch.io/repo/com.google-code.libphonenumber/libphonenumber/4.2>), you could upload the file similarly to the examples above in the bucket named `javalib`. The corresponding local bucket path would look like the following: `/buckets/bfs-default/javalib/libphonenumber-4.2.jar`.

In the script below you can see how this path is specified to be able to import the library.

```
CREATE JAVA SCALAR SCRIPT jphone(num VARCHAR(2000))
RETURNS VARCHAR(2000) AS
%jar /buckets/bfsdefault/javalib/libphonenumber-4.2.jar;

import com.google.i18n.phonenumbers.PhoneNumberUtil;
import com.google.i18n.phonenumbers.NumberParseException;
import com.google.i18n.phonenumbers.Phonenumber.PhoneNumber;

class JPHONE {
    static String run(ExaMetadata exa, ExaIterator ctx) throws Exception {
        PhoneNumberUtil phoneUtil = PhoneNumberUtil.getInstance();
        try {
            PhoneNumber swissNumberProto = phoneUtil.parse(ctx.getString("num"),
                    "DE");
            return swissNumberProto.toString();
        } catch (NumberParseException e) {
            System.err.println("NumberParseException thrown: " + e.toString());
        }
        return "failed";
    }
}
```

Python libraries

Python libraries are often provided in the form of .whl files. You can easily integrate such libraries into Exasol by uploading the file to a bucket and extend the Python search path appropriately.

If you for instance want to use Google's library for processing phone numbers (<https://pypi.python.org/pypi/phonenumbers>), you could upload the file into the bucket named `pylib`. The corresponding local path would look like the following: `/buckets/bfsdefault/pylib/phonenumbers-7.7.5-py2.py3-none-any.whl`.

In the script below you can see how this path is specified to be able to import the library.

```
CREATE PYTHON SCALAR SCRIPT phonenumbers(phone_number VARCHAR(2000))
RETURNS VARCHAR(2000) AS
```

```

import sys
import glob

sys.path.extend(glob.glob('/buckets/bfsdefault/pylib/*'))
import phonenumbers

def run(c):
    return str(phonenumbers.parse(c.phone_number, None))
/

SELECT phononenumber('+1-555-2368') AS ghost_busters;

GHOST_BUSTERS
-----
Country Code: 1 National Number: 55552368

```

R packages

The installation of R packages is a bit more complex because they have to be compiled using the c compiler. To manage that, you can download the existing Exasol Linux container, compile the package in that container and upload the resulting package into BucketFS. Details about the Linux container will be explained in the following chapter.

A simple method is using Docker as described in the following example.

```

$> bucketfs = http://192.168.6.75:1234
$> cont = /default/EXAClusterOS/ScriptLanguages-2016-10-21.tar.gz
$> docker $bucketfs$cont import my_dock
$> mkdir r_pkg
$> docker run -v `pwd`/r_pkg:/r_pkg --name=my_dock -it my_dock /bin/bash

```

Again we want to use an existing package for processing phone numbers, this time from Steve Myles (<https://cran.r-project.org/web/packages/phonenumbers/index.html>).

Within the Docker container, you can start R and install it:

```

# export R_LIBS="/r_pkg/"
# R
> install.packages('phonenumbers', repos="http://cran.r-project.org")
Installing package into '/r_pkg'
(as 'lib' is unspecified)
trying URL 'http://cran.r-project.org/src/contrib/phonenumbers_0.2.2.tar.gz'
Content type 'application/x-gzip' length 10516 bytes (10 KB)
=====
downloaded 10 KB

```

The first line installs the package from the Linux container into the subfolder `r_pkg` which can be accessed outside the Docker container.

Afterwards, the resulting tgz archive is uploaded into the bucket named `rlib`:

```

$> bucketfs = http://w:writepw@192.168.6.75:1234
$> curl -vX PUT -T r_pkg.tgz $bucketfs/rlib/r_pkg.tgz

```

In the following script you can see how the resulting local path (/buckets/bfsdefault/rlib/r_pkg) is specified to be able to use the library.

```

CREATE R SCALAR SCRIPT tophone(letters VARCHAR(2000)) RETURNS INT AS
.libPaths( c( .libPaths(), "/buckets/bfsdefault/rlib/r_pkg" ) )
library(phonenumber)

run <- function(ctx) {
  letterToNumber(ctx$letters, qz = 1)
}
/

```

Installing new script languages

Due to Exasol's open script framework it is simply possible to integrate completely new script languages into Exasol, following these 3 steps:

1. Creating a functioning language client
2. Upload the resulting client into a bucket
3. Define a script language alias ([ALTER SESSION](#) or [ALTER SYSTEM](#))

How you create a language client will be explained in the next chapter. In principle, the language has to be expanded by some small APIs which implement the communication between Exasol and the language. Afterwards the client is compiled for the usage in the pre-installed Exasol Linux Container, so that it can be started within a secure process on the Exasol cluster.

Uploading and providing files through the BucketFS has already been explained in the previous chapters.

The last step creates a link between the language client and Exasol's SQL language, actually the [CREATE SCRIPT](#) command. This facilitates many options to try out new versions of a language or completely new languages and finally replace them completely.

If you for instance plan to migrate from Python 2 to Python 3, you could upload a new client and link the alias PYTHON temporarily to the new version via [ALTER SESSION](#). After a thorough testing phase, you can finally switch to the new version for all users via the [ALTER SYSTEM](#) statement.

On the other side, it is also possible to use both language versions in parallel by defining two aliases separately (e.g. PYTHON2 and PYTHON3).

Creating a script client

The main task of installing new languages is developing the script client. If you are interested in a certain language, you should first check whether the corresponding client has already been published in our open source repository (see <https://github.com/exasol/script-languages>). Otherwise we would be very happy if you would contribute self-developed clients to our open source community.

A script client is based on a Linux container which is stored in BucketFS. The pre-installed script client for languages Python, Java and R is located in the default bucket of the default BucketFS. Using the Linux container technology, an encapsulated virtual machine is started in a safe environment whenever an SQL query contains script code. The Linux container includes a complete Linux distribution and starts the corresponding script client for executing the script code. In general, you could also upload your own Linux container and combine it with your script client.

The script client has to implement a certain protocol that controls the communication between the scripts and the database. For that purpose, the established technologies ZeroMQ™ (see <http://zeromq.org>) and Google's Protocol Buffers™ (<https://github.com/google/protobuf>) are used. Because of the length, the actual protocol definition is not included in this user manual. For details, please have a look into our open source repository (see <https://github.com/exasol/script-languages>) where you'll find the following:

- Protocol specification
- Necessary files for building the Exasol Linux container (Docker configuration file and build script)

- Example implementations for further script clients (e.g. C++ and a native Python client that supports both Python 2 and Python 3)

Script aliases for the integration into SQL

After building and uploading a script client, you have to configure an alias within Exasol. The database then knows where each script language has been installed.

You can change the script aliases via the commands [ALTER SESSION](#) and [ALTER SYSTEM](#), either session-wide or system-wide. This is handy especially for using several language versions in parallel, or migrating from one version to another.

The current session and system parameters can be found in [EXA_PARAMETERS](#). The scripts aliases are defined via the parameter `SCRIPT_LANGUAGES`:

```
SELECT session_value FROM exa_parameters
WHERE parameter_name='SCRIPT_LANGUAGES';

PARAMETER_NAME
-----
PYTHON=builtin_python R=builtin_r JAVA=builtin_java
```

These values are not very meaningful since they are just internal macros to make that parameter compact and to dynamically use the last installed version. Written out, the alias for Java would look like the following:

```
JAVA=localzmq+protobuf:///bfsdefault/default/EXAClusterOS/ScriptLanguages-2016-10-21/?lang=java#buckets/bfsdefault/default/EXASolution-6.0.0/exaudfclient
```

That alias means that for all `CREATE JAVA SCRIPT` statements, the Exasol database will use the script client `exaudfclient` from local path `buckets/bfsdefault/default/EXASolution-2016-10-21`, started within the Exasol Linux container from path `bfsdefault/default/EXAClusterOS/ScriptLanguages-2016-10-21`. The used communication protocol is `localzmq+protobuf` (this is the only supported protocol so far).

For the three pre-installed languages (Python, R, Java), Exasol uses one single script client which evaluates the parameter `lang=java` to differentiate between these languages. That's why the internal macro for the Python alias looks nearly identical. Script clients implemented by users can of course define and evaluate such optional parameters individually.

In general, a script alias contains the following elements:

```
<alias>=localzmq+protobuf://<path_to_linux-container>/[?<client_param_list>]#<path_to_client>
```

 Please note that a script alias may not contain spaces

Maybe you have noticed in the Java alias that the path of the Linux container begins with the BucketFS while the client path contains the prefix `buckets/`. The reason for that is that the client is started in a secure environment within the Linux container and may only access the existing (and visible) buckets. Access is granted just to the embedded buckets (via `mount`), but not to the real server's underlying file system. The path `/buckets/` can be used read-only from within scripts as described in the previous chapter.

You can define or rename any number of aliases. As mentioned before, we recommend to test such adjustments at first in your own session ([ALTER SESSION](#)) before making that change globally visible via [ALTER SYSTEM](#).

3.7. Virtual schemas

3.7.1. Virtual schemas and tables

Virtual schemas provide a powerful abstraction to conveniently access arbitrary data sources. Virtual schemas are a kind of read-only link to an external source and contain virtual tables which look like regular tables except that the actual data are not stored locally.

 Please note that virtual schemas are part of the Advanced Edition of Exasol.

After creating a virtual schema, its included tables can be used in SQL queries and even combined with persistent tables stored directly in Exasol, or with other virtual tables from other virtual schemas. The SQL optimizer internally translates the virtual objects into connections to the underlying systems and implicitly transfers the necessary data. SQL conditions are tried to be pushed down to the data sources to ensure minimal data transfer and optimal performance.

That's why this concept creates a kind of logical view on top of several data sources which could be databases or other data services. By that, you can either implement a harmonized access layer for your reporting tools. Or you can use this technology for agile and flexible ETL processing, since you don't need to change anything in Exasol if you change or extend the objects in the underlying system.

The following basic example shows you how easy it is to create and use a virtual schema by using our JDBC adapter to connect Exasol with a Hive system.

```
-- Create the schema by specifying some adapter access properties
CREATE VIRTUAL SCHEMA hive
  USING adapter.jdbc_adapter
  WITH
    CONNECTION_STRING = 'jdbc:hive://myhost:21050;AuthMech=0'
    USERNAME         = 'hive-user'
    PASSWORD         = 'secret-password'
    SCHEMA_NAME      = 'default';

-- Explore the tables in the virtual schema
OPEN SCHEMA hive;
SELECT * FROM cat;
DESCRIBE clicks;

-- Run queries against the virtual tables
SELECT count(*) FROM clicks;
SELECT DISTINCT USER_ID FROM clicks;
-- queries can combine virtual and native tables
SELECT * from clicks JOIN native_schema.users ON clicks.userid = users.id;
```

SQL Commands to manage virtual schemas:

CREATE VIRTUAL SCHEMA Creating a virtual schema (see also [CREATE SCHEMA](#))

DROP VIRTUAL SCHEMA Deleting a virtual schema and all contained virtual tables (see also [DROP SCHEMA](#))

ALTER VIRTUAL SCHEMA Adjust the properties of an virtual schema or refresh the metadata using the REFRESH option (see also [ALTER SCHEMA](#))

EXPLAIN VIRTUAL Useful to analyze which resulting queries for external systems are created by the Exasol compiler (see also [EXPLAIN VIRTUAL](#))

Instead of shipping just a certain number of supported connectors (so-called adapters) to other technologies, we decided to provide users an open, extensible framework where the connectivity logic is shared as open source. By that, you can easily use existing adapters, optimize them for your need or even create additional adapters to all kinds of data sources by your own without any need to wait for a new release from Exasol.

In the following chapters, we will explain how that framework works and where you'll find the existing adapters.

3.7.2. Adapters and properties

If you create a virtual schema, you have to specify the corresponding adapter - to be precise an adapter script - which implements the logic how to access data from the external system. This varies from technology to technology, but always includes two main task:

Read metadata	Receive information about the objects included in the schema (tables, columns, types) and define the logic how to map the data types of the source system to the Exasol data types.
Push down query	Push down parts of the Exasol SQL query into an appropriate query the the external system. The adapter defines what kind of logic Exasol can push down (e.g. filters or certain functions). The Exasol optimizer will then try to push down as much as possible and execute the rest of the query locally on the Exasol cluster.

Adapters are similar to UDF scripts (see also [Section 3.6, “UDF scripts”](#) and the details in the next chapter). They can be implemented in one of the supported programming languages, for example Java or Python, and they can access the same metadata which is available within UDF scripts. To install an adapter you simply download and execute the SQL scripts which creates the adapter script in one of your normal schemas.



The existing open source adapters provided by Exasol can be found in our GitHub repository: <https://www.github.com/exasol/virtual-schemas>



A very generic implementation is our JDBC adapter with which you can integrate nearly any data source providing a Linux JDBC driver. For some database systems, an appropriate dialect was already implemented to push as much processing as possible down to the underlying system. Please note that for using this JDBC adapter you have to upload the corresponding in BucketFS for the access from adapter scripts (see also [Section 3.6.4, “The synchronous cluster file system BucketFS”](#)). Additionally the driver has to be installed via EXAoperation, because the JDBC adapter executes an implicit IMPORT command).

SQL Commands to manage adapter scripts:

CREATE ADAPTER SCRIPT Creating an adapter script, see [CREATE SCRIPT](#)

DROP ADAPTER SCRIPT Deleting an adapter script, see [DROP SCRIPT](#)

EXPLAIN VIRTUAL Useful to analyze which resulting queries for external systems are created by the Exasol compiler, see also [EXPLAIN VIRTUAL](#)

The following example shows a shortened adapter script written in Python.

```
CREATE SCHEMA adapter;
CREATE OR REPLACE PYTHON ADAPTER SCRIPT adapter.my_adapter AS
def adapter_call(request):
    # Implement your adapter here.
    # ...
    # It has to return an appropriate response.
/
```

Afterwards you can create virtual schemas by providing certain properties which are required for the adapter script (see initial example). These properties typically define the necessary information to establish a connection to the external system. In the example, this was the jdbc connection string and the credentials.

But properties can flexibly defined and hence contain all kinds of auxiliary data which controls the logic how to use the data source. If you implement or adjust your own adapter scripts, then you can define your own properties and use them appropriately.

The list of specified properties of a specific virtual schema can be seen in the system table **EXA_ALL_VIRTUAL_SCHEMA_PROPERTIES**. After creating the schema, you can adjust these properties using the SQL command **ALTER SCHEMA**.

After the virtual schema was created in the described way, you can use the contained tables in the same way as the normal tables in Exasol, and even combine them in SQL queries. And if you want to use this technology just for simple ETL jobs, you can of course simply materialize a query on the virtual tables:

```
CREATE VIRTUAL SCHEMA hive
  USING adapter.jdbc_adapter
  WITH
    CONNECTION_STRING = 'jdbc:hive://myhost:21050;AuthMech=0'
    USERNAME          = 'hive-user'
    PASSWORD          = 'secret-password'
    SCHEMA_NAME       = 'default';

CREATE TABLE my_schema.clicks_copy FROM
  (SELECT * FROM hive.clicks);
```

3.7.3. Grant access on virtual tables

The access to virtual data works similar to the creation of view by simply granting the **SELECT** privilege for the virtual schema. In the case of a virtual schema you can grant this right only for the schema altogether (via **GRANT SELECT ON SCHEMA**). Alternatively, the user is the owner of the schema.

Internally, this works similar to views since the check for privileges is executed in the name of the script owner. By that the details are completely encapsulated, i.e. the access to adapter scripts and the credentials to the external system.

Limit access to specific tables

If you don't want to grant full access for a virtual schema but selectively for certain tables, you can do that by different alternatives:

Views	Instead of granting direct access to the virtual schema you can also create views on that data and provide indirect access for certain users.
Logic within the adapter script	It is possible to solve this requirement directly within the adapter script. E.g. in our published JDBC adapter, there exists the parameter TABLE_FILTER through which you can define a list of tables which should be visible (see https://www.github.com/exasol). If this virtual schema property is not defined, then all available tables are made visible.

3.7.4. Privileges for administration

The privileges needed to create and administrate the adapter scripts and virtual schemas can be found in the corresponding SQL commands **CREATE SCHEMA**, **ALTER SCHEMA**, **DROP SCHEMA** and **CREATE SCRIPT** or in the overview in [Appendix B, Details on rights management](#).

Encapsulate access credentials via connections

If you take a look at the introductory example you will notice that the credentials for the external system are defined in plain text as parameters of the virtual schema. This method is indeed very easy but not acceptable in most cases, because this information can afterwards be found in system tables. That's why we always recommend to use connection objects for encapsulating this security-relevant information.

In this case you only define the connection name but not the actual credentials:

```
CREATE CONNECTION hive_conn
  TO 'jdbc:hive://myhost:21050;AuthMech=0'
  USER 'username'
  IDENTIFIED BY 'secret-password';

CREATE VIRTUAL SCHEMA vs_hive
  USING adapter.jdbc_adapter
  WITH
    SQL_DIALECT = 'HIVE'
    CONNECTION_NAME = 'HIVE_CONN'
    SCHEMA_NAME = 'default';
```

Of course, the adapter script has to support this and extract the credentials from the connection to be able to establish a connection to the external system.

The administrator of the virtual schema needs the following privileges to enable the encapsulation via connections:

1. You have to grant the connection itself to the administrator (GRANT CONNECTION), because in most cases, an adapter script will internally generate an IMPORT statement in the two-phased execution (see also [Section 3.7.6, “Details for experts”](#)) which uses this connection like standard IMPORT statements do. Since IMPORT is an internal Exasol command and not a public script, the credentials cannot be extracted at all.
2. In most cases you also need access to the connection details (actually the *user* and *password*), because the adapter script needs a direct connection to read the metadata from the external in case of commands such as CREATE and REFRESH. For that purpose the special ACCESS privilege has been introduced, because of the criticality of these data protection relevant connection details. By the statement GRANT ACCESS ON CONNECTION [FOR SCRIPT] you can also limit that access only to a specific script (FOR SCRIPT clause) and ensure that the administrator cannot access that data himself (e.g. by creating a new script which extracts and simply returns the credentials). Of course that user should only be allowed to execute, but not alter the script by any means.

In the example below, the administrator gets the appropriate privileges to create a virtual schema by using the adapter script (`jdbc_adapter`) and a certain connection (`hive_conn`).

```
GRANT CREATE VIRTUAL SCHEMA TO user_hive_access;
GRANT EXECUTE ON SCRIPT adapter.jdbc_adapter TO user_hive_access;
GRANT CONNECTION hive_conn TO user_hive_access;
GRANT ACCESS ON CONNECTION hive_conn
  FOR SCRIPT adapter.jdbc_adapter TO user_hive_access;
```

3.7.5. Metadata

You'll find detailed information about all created adapter scripts and virtual schemas in the following system tables:

Virtual schemas

- [EXA_VIRTUAL_SCHEMAS](#)
- [EXA_ALL_VIRTUAL_SCHEMA_PROPERTIES](#)

- EXA_USER_VIRTUAL_SCHEMA_PROPERTIES
- EXA_DBA_VIRTUAL_SCHEMA_PROPERTIES
- EXA_ALL_VIRTUAL_TABLES
- EXA_DBA_VIRTUAL_TABLES
- EXA_USER_VIRTUAL_TABLES
- EXA_ALL_VIRTUAL_COLUMNS
- EXA_DBA_VIRTUAL_COLUMNS
- EXA_USER_VIRTUAL_COLUMNS

Adapter scripts

- EXA_ALL_SCRIPTS (SCRIPT_TYPE='ADAPTER')
- EXA_DBA_SCRIPTS (SCRIPT_TYPE='ADAPTER')
- EXA_USER_SCRIPTS (SCRIPT_TYPE='ADAPTER')

Restricted ACCESS rights for connections

- EXA_DBA_RESTRICTED_OBJ_PRIVS
- EXA_ROLE_RESTRICTED_OBJ_PRIVS
- EXA_USER_RESTRICTED_OBJ_PRIVS

Connections

- EXA_ALL_CONNECTIONS
- EXA_DBA_CONNECTIONS
- EXA_SESSION_CONNECTIONS
- EXA_DBA_CONNECTION_PRIVS
- EXA_USER_CONNECTION_PRIVS

3.7.6. Details for experts

If you just want to use existing adapters to create virtual schemas, then the previous sections should have provided enough information to understand:

1. How to download the necessary adapter script
2. How to implement the adapter script in your Exasol database
3. How to create and use a virtual schema

But if you want to know more about the underlying concepts, or if you even plan to create or adjust your own adapter, then please read on.

Every time you access data of a virtual schema, on one node of the cluster a container of the corresponding language is started, e.g. a JVM or a Python container. Inside this container, the code of the adapter script will be loaded. Exasol interacts with the adapter script using a simple request-response protocol encoded in JSON. The database takes the active part sending the requests by invoking a callback method.

In case of Python, this method is called `adapter_call` per convention, but this can vary.

Let's take a very easy example of accessing a virtual table.

```
SELECT name FROM my_virtual_schema.users WHERE name like 'A%';
```

The following happens behind the scenes:

1. Exasol determines that a virtual table is involved, looks up the corresponding adapter script and starts the language container on one single node in the Exasol cluster.

2. Exasol sends a request to the adapter, asking for the capabilities of the adapter.
3. The adapter returns a response including the supported capabilities, for example whether it supports specific WHERE clause filters or specific scalar functions.
4. Exasol sends an appropriate pushdown request by considering the specific adapter capabilities. For example, the information for column projections (in the example above, only the single column name is necessary) or filter conditions is included.
5. The adapter processes this request and sends back a certain SQL query in Exasol syntax which will be executed afterwards. This query is typically an IMPORT statement or a SELECT statement including an row-emitting UDF script which cares about the data processing.

The example above could be transformed into these two alternatives (IMPORT and SELECT):

```
SELECT name FROM ( IMPORT FROM JDBC AT ... STATEMENT 'SELECT name from remoteschema.users WHERE name LIKE "A%"' );
```

```
SELECT name FROM ( SELECT GET_FROM_MY_DATA_SOURCE('data-source-address', 'required_column=name') ) WHERE name LIKE 'A%';
```

In the first alternative, the adapter can handle filter conditions and creates an IMPORT command including a statement which is sent to the external system. In the second alternative, a UDF script is used with two parameters handing over the address of the data source and the column projection, but not using any logic for the filter condition. This would then be processed by Exasol rather than the data source.

Please be aware that the examples show the fully transformed query while only the inner statements are created by the adapter.

6. The received data is directly integrated into the overall query execution of Exasol.

To understand the full API of the adapter scripts we refer to our open source repository (<https://www.github.com/exasol>) and our existing adapters. They include API documentation and it is far easier to read these concrete examples than to explain all details in this user manual.

If you want to enhance existing or create completely new adapters, we recommend to use [EXPLAIN VIRTUAL](#) to easily analyze what the adapter is exactly pushing down to the underlying system.

3.8. SQL Preprocessor

3.8.1. How does the SQL Preprocessor work?

In Exasol a SQL Preprocessor is available which can preprocess all executed SQL statements. By this means, unsupported SQL constructs can be transformed into existing SQL features (see examples below). Additionally, you can easily introduce syntactic sugar by replacing simple constructs with more complex elements.



If you want to use the SQL Preprocessor, please read this chapter carefully since the impacts on your database system could be extensive.

The SQL Preprocessor is deactivated by default. Via the statements [ALTER SESSION](#) and [ALTER SYSTEM](#), you can define a script session or system wide which is responsible for the preprocessing of all SQL commands. Before a SQL statement is passed to the actual database compiler, the preprocessor does a kind of text transformation. Within the script, you can get and set the original text and manipulate it by using our auxiliary library (see next section). Details to the scripting language can be found in [Section 3.5, “Scripting”](#).

The SQL Preprocessor is a powerful tool to flexibly extend the SQL language of Exasol. But you should also be very careful before activating such a SQL manipulation system wide. In worst case no single SQL statement could work anymore. But you can always deactivate the preprocessing via [ALTER SESSION](#) and [ALTER SYSTEM](#) (by setting the parameter `SQL_PREPROCESSOR_SCRIPT` to the `NULL` value), because these statements are deliberately excluded from the preprocessing. For data security reasons, we also excluded all statements which include passwords ([CREATE USER](#), [ALTER USER](#), [CREATE CONNECTION](#), [ALTER CONNECTION](#), [IMPORT](#), [EXPORT](#) if the IDENTIFIED BY clause was specified).

In the auditing table [EXA_DB_AUDIT_SQL](#), a separate entry for the execution of the preprocessor script is added (EXECUTE SCRIPT and the original text within a comment). The executed transformed SQL statement is listed in another entry.

3.8.2. Library sqlparsing

For an easy to use SQL text manipulation, we included a special library with certain parsing capabilities.

Overview

Split into tokens	<ul style="list-style-type: none">• <code>tokenize()</code>
Identification of token types	<ul style="list-style-type: none">• <code>iswhitespace()</code>• <code>iscomment()</code>• <code>iswhitespaceorcomment()</code>• <code>isidentifier()</code>• <code>iskeyword()</code>• <code>isstringliteral()</code>• <code>isnumericliteral()</code>• <code>isany()</code>
Normalizing a string	<ul style="list-style-type: none">• <code>normalize()</code>
Finding token sequences	<ul style="list-style-type: none">• <code>find()</code>
Access to the SQL text	<ul style="list-style-type: none">• <code>getsqltext()</code>• <code>setsqltext()</code>

Details

- **sqlparsing.tokenize(sqlstring)**

Splits an input string into an array of strings which correspond to the tokens recognized by the database compiler. If you concatenate these tokens, you will get exactly the original input string (including upper/lower case, line breaks, whitespaces, etc.). Hence, the equation `table.concat(tokens) == sqlstring` is valid.

The following tokens are possible:

- Valid SQL identifiers, e.g. `test.tab` or `"foo"."bar"`
- Keywords, e.g. `SELECT`
- String literals, e.g. `'abcdef'`
- Numerical literals without sign, e.g. `123e4`
- Connected whitespaces corresponding to the SQL standard
- Comments corresponding to the SQL standard (line and block comments)
- Multi character tokens, e.g. `'::', '||', '>', '>>', '<<', '>=', '<='`, `'<>'`, `'!=', '^='`
- Single character tokens, e.g. `'+', '!', '/', '*', '~'`

```
CREATE SCRIPT example(sql_text) AS
    local tokens = sqlparsing.tokenize(sql_text)
    for i=1,#tokens do
        print(tokens[i])
    end
/
EXECUTE SCRIPT example('SELECT dummy FROM dual') WITH OUTPUT;
OUTPUT
-----
SELECT
dummy
FROM
dual
```

- **sqlparsing.iscomment(tokenstring)**

Returns whether the input string is a comment token.

- **sqlparsing.iswhitespace(tokenstring)**

Returns whether the input string is a whitespace token.

- **sqlparsing.iswhitespaceorcomment(tokenstring)**

Returns whether the input string is a whitespace or comment token. This function can be useful for function `find()`, because it filters all irrelevant tokens corresponding to the SQL standard.

- **sqlparsing.isidentifier(tokenstring)**

Returns whether the input string is an identifier token.

- **sqlparsing.iskeyword(tokenstring)**

Returns whether the input string is a SQL keyword token (e.g. `SELECT`, `FROM`, `TABLE`). The functions `isidentifier()` and `iskeyword()` return both `true` for non-reserved keywords. Hence, you are also able to identify non-reserved keywords.

- **sqlparsing.isstringliteral(tokenstring)**

Returns whether the input string is a string literal token.

- **sqlparsing.isnumericliteral(tokenstring)**

Returns whether the input string is a numeric literal token.

- **sqlparsing.isany(tokenstring)**

Always returns `true`. This can e.g. be useful if you want to find any first relevant token (as match function within the method `find()`).

- **sqlparsing.normalize(tokenstring)**

Returns a normalized string for similar representations (e.g. upper/lower case identifiers), on the basis of the following rules:

- Regular identifiers are transformed into upper-case letters, e.g. `dual` -> `DUAL`
- Keywords are transformed in upper-case letters, e.g. `From` -> `FROM`
- Whitespace-Tokens of any size are replaced by a single whitespace
- In numerical literals, an optional lower-case 'e' is replaced by 'E', e.g. `1.2e34` -> `1.2E34`

- **sqlparsing.find(tokenlist, startTokenNr, searchForward, searchSameLevel, ignoreFunction, match1, [match2, ... matchN])**

Searches in the token list, starting from positions `startTokenNr`, forward or backward (`searchForward`) and optionally only within the current level of brackets (`searchSameLevel`), for the directly successive sequence of tokens which are matched by parameters `match1`, ... `matchN`. In that search process, all tokens which match by function `ignoreFunction` will be not considered by the match functions.

If the searched token sequence is found, then an array of size `N` is returned (in case of `N` match elements) whose `X`-th entry contains the position of the token within the token list which was matched by `matchX`. If the token sequence was not found, the function returns `nil`.

Details on parameters:

`tokenlist` List of tokens which is e.g. produced by the function `tokenize`

`startTokenNr` Number of the first token to be considered for the search

`searchForward` Defines whether the search should be applied forward (`true`) or backward (`false`). This affects only the direction by that the search process is moving across the list, but the match functions always search forward.

That means that if you e.g. search the token sequence `KEYWORD`, `IDENTIFIER`, within the token list `'select' 'abc' 'from' 'dual'`, and start from position 3, then `'from'` `'dual'` will be matched and not `'from' 'abc'`, even when searching backward. If you start your search at position 2, then the backward search will return `'select' 'abc'`, and the forward search will return `'from' 'dual'`.

`searchSameLevel` Defines whether the search should be limited to the current level of brackets (`true`) or also beyond (`false`). This applies only to the match of the first token of the sequence. Subsequent tokens can also be located in more inner bracket levels. That means that the search of the token sequence `'=' '(' 'SELECT'` is also possible if it is constrained to the current level, although the `'SELECT'` is located in the next inner bracket level. The option `searchSameLevel` is especially useful for finding the corresponding closing bracket, e.g. of a subquery.

Example: Search the closing bracket within the token sequence `'SELECT' 't1.x' '+' '(' 'SELECT' 'min' '(' 'y' ')' 'FROM' 't2' ')' 'FROM' 't1'` which corresponds to the bracket

at position 4: `sqlparsing.find(tokens, 4, true, true, sqlparsing.iswhitespaceorcomment, '')`.

ignoreFunction

Here, a function of type `function(string)->bool` is expected. The tokens for which the function `ignoreFunction` returns `true` will be ignored by the match functions. That means in particular that you can specify tokens types which may occur within the sequence without breaking the match. In many cases, the function `iswhitespaceorcomment` is useful for that purpose.

match1...matchN

By `match1..matchN`, the searched token sequence is specified. These parameters should either be functions of type `function(tokenstring)->bool` or simple strings. A token sequence is searched where the first token matches `match1`, the second matches `match2`, etc., while tokens in between are ignored if function `ignoreFunction` returns `true`. If a parameter is a string, then the comparison `normalize(tokenstring)==normalize(matchstring)` is applied.

- **sqlparsing.getsqltext()**

Returns the current SQL statement text.



This function is only available within the main SQL Preprocessor script.

- **sqlparsing.setsqltext(string)**

Sets the SQL statement text to a new value which will eventually be passed to the database compiler for execution.



This function is only available within the main SQL Preprocessor script.

3.8.3. Best Practice

To achieve a smooth development and global activation of the SQL Preprocessor, you should consider the following recommendations:

- You should extensively test a preprocessor script in your own session before activating it system wide.
- The SQL processing should be implemented by the use of separate auxiliary scripts and integrated in one main script which is just a wrapper to hand over the SQL text (e.g. `sqlparsing.setsqltext(myscript.preprocess(sqlparsing.getsqltext()))`). The reason for that approach is that the functions `getsqltext()` and `setsqltext()` are only available within the preprocessing and not in normal script executions. By the separation you can test the processing on several test SQL constructs (e.g. on your own daily SQL history stored within a table) before activating the main script as preprocessor script.
- Be sure that all necessary privileges are granted to execute the preprocessor script. It is recommended that you start a test with a user without special rights. Otherwise, certain user groups could be blocked of executing any SQL statements.
- The preprocessing should be as simple as possible. Especially `query()` and `pquery()` should only be used in exceptional cases if you activate the preprocessing globally, because all SQL queries will be decelerated, and a parallel access on similar tables increases the risk of transaction conflicts.

3.8.4. Examples

In the following you find some examples for Preprocessor scripts which shall show you the functionality and power of this feature.

Example 1: IF() function

In this example the IF() function, currently not supported in Exasol, is transformed into an equivalent CASE-WHEN expression.

```

CREATE SCRIPT transformIf() AS
function processIf(sqltext)
    while (true) do
        local tokens = sqlparsing.tokenize(sqltext)
        local ifStart = sqlparsing.find(tokens,
                                         1,
                                         true,
                                         false,
                                         sqlparsing.iswhitespaceorcomment,
                                         'IF',
                                         '(')

        if (ifStart==nil) then
            break;
        end
        local ifEnd = sqlparsing.find(tokens,
                                       ifStart[2],
                                       true,
                                       true,
                                       sqlparsing.iswhitespaceorcomment,
                                       ')')

        if (ifEnd==nil) then
            error("if statement not ended properly")
            break;
        end
        local commas1 = sqlparsing.find(tokens,
                                         ifStart[2]+1,
                                         true,
                                         true,
                                         sqlparsing.iswhitespaceorcomment,
                                         ',')

        if (commas1==nil) then
            error("invalid if function")
            break;
        end
        local commas2 = sqlparsing.find(tokens,
                                         commas1[1]+1,
                                         true,
                                         true,
                                         sqlparsing.iswhitespaceorcomment,
                                         ',')

        if (commas2==nil) then
            error("invalid if function")
            break;
        end
        local ifParam1=table.concat(tokens, '', ifStart[2]+1, commas1[1]-1)
        local ifParam2=table.concat(tokens, '', commas1[1]+1, commas2[1]-1)
        local ifParam3=table.concat(tokens, '', commas2[1]+1, ifEnd[1]-1)
        local caseStmt='CASE WHEN ('..ifParam1..'') != 0 \
                        THEN ('..ifParam2..'') \
                        ELSE ('..ifParam3..'') END '
        sqltext=table.concat(tokens, '',1,
                             ifStart[1]-1)..caseStmt..table.concat(tokens,

```

```

        '',
        ifEnd[1]+1)

    end
    return sqltext
end
/

CREATE SCRIPT sql_preprocessing.preprocessIf() AS
  import( 'sql_preprocessing.transformIf', 'transformIf' )
  sqlparsing.setsqltext(
    transformIf.processIf(sqlparsing.getsqltext()))
/

SELECT IF( 3+4 > 5, 6, 7 ) from dual;
Error: [42000] syntax error, unexpected IF_ [line 1, column 8]

ALTER SESSION SET sql_preprocessor_script=
  sql_preprocessing.preprocessIf;
SELECT IF( 3+4 > 5, 6, 7 ) AS coll FROM dual;

COL1
-----
  6

```

Example 2: ls command

In this example, the familiar Unix command **ls** is transferred to the database world. This command returns either the list of all objects within a schema or the list of all schemas if no schema is opened. Additionally, you can apply filters (case insensitive) like e.g. **ls '%name%'** to display all objects whose name contains the text 'name'.

```

CREATE SCRIPT sql_preprocessing.addunixcommands() AS
  function processLS(input, tokens, commandPos)
    local result = query("SELECT CURRENT_SCHEMA")
    local current_schema = result[1][1]
    local returnText = ""
    local searchCol = ""
    if (current_schema==null) then
      returnText = "SELECT schema_name FROM exa_schemas WHERE true"
      searchCol = "schema_name"
    elseif (current_schema=='SYS' or current_schema=='EXA_STATISTICS') then
      returnText = "SELECT object_name, object_type FROM exa_syscat \
                  WHERE schema_name='..current_schema..''"
      searchCol = "object_name"
    else
      returnText = "SELECT object_name, object_type FROM exa_all_objects \
                  WHERE root_type='SCHEMA' \
                  AND root_name='..current_schema..''"
      searchCol = "object_name"
    end
    local addFilters = {}
    local lastValid = commandPos
    local foundPos = sqlparsing.find(tokens,
                                      lastValid+1,
                                      true,
                                      false,
                                      sqlparsing.iswhitespaceorcomment,

```

```

sqlparsing.isany)
while (not(foundPos==nil) )
do
    local foundToken = tokens[foundPos[1]]
    if (sqlparsing.isstringliteral(foundToken)) then
        addFilters[#addFilters+1] = "UPPER(..searchCol..) \
                                LIKE UPPER(..foundToken .. ))"
    elseif (not (sqlparsing.normalize(foundToken) == ';')) then
        error("only string literals allowed as arguments for ls,\n\
              but found '\"..foundToken..\"''")
    end
    lastValid = foundPos[1]
    foundPos = sqlparsing.find(tokens,
                               lastValid+1,
                               true,
                               false,
                               sqlparsing.iswhitespaceorcomment,
                               sqlparsing.isany)
end
if ( #addFilters > 0 ) then
    local filterText = table.concat(addFilters, " OR ")
    return returnText.." AND ("..filterText.."").." ORDER BY "..searchCol
else
    return returnText.." ORDER BY "..searchCol
end
end

function processUnixCommands(input)
    local tokens = sqlparsing.tokenize(input)
    local findResult = sqlparsing.find(tokens,
                                       1,
                                       true,
                                       false,
                                       sqlparsing.iswhitespaceorcomment,
                                       sqlparsing.isany)
    if (findResult==nil) then
        return input
    end
    local command = tokens[findResult[1]]
    if (sqlparsing.normalize( command )=='LS') then
        return processLS(input, tokens, findResult[1])
    end
    return input;
end
/
CREATE SCRIPT sql_preprocessing.preprocessWithUnixTools() AS
import( 'sql_preprocessing.addunixcommands', 'unixCommands' )
sqlparsing.setsqltext(
    unixCommands.processUnixCommands(sqlparsing.getsqltext()));
/
ALTER SESSION SET sql_preprocessor_script=
    sql_preprocessing.preprocessWithUnixTools;

OPEN SCHEMA sql_preprocessing;
LS '%unix%';

```

OBJECT_NAME	OBJECT_TYPE
ADDUNIXCOMMANDS	SCRIPT
PREPROCESSWITHUNIXTOOLS	SCRIPT
CLOSE SCHEMA;	
LS;	
SCHEMA_NAME	
SCHEMA_1	
SCHEMA_2	
SCHEMA_3	
SQL_PREPROCESSING	

Example 3: ANY/ALL

ANY and ALL are SQL constructs which are currently not supported by Exasol. By the use of the following script, you can add this functionality.

```
CREATE SCRIPT sql_preprocessing.transformAnyAll() AS
    function rebuildAnyAll(inputsqltext)
        local sqltext = inputsqltext;
        local tokens = sqlparsing.tokenize(sqltext);
        local found = true;
        local searchStart = 1;
        -- search for sequence >|=|<|= ANY|ALL ( SELECT
        repeat
            local foundPositions =
                sqlparsing.find(tokens,
                    searchStart,
                    true,
                    false,
                    sqlparsing.iswhitespaceorcomment,
                    function (token)
                        return (token=='<' or token=='<=' or token=='>' or token=='>=' or token=='!=')
                            or token=='=' );
                    end,      -- match <|=|>|=|=|!=|>
                    function ( token )
                        local normToken = sqlparsing.normalize(token);
                        return (normToken=='ANY' or normToken=='SOME'
                            or normToken=='ALL');
                    end,      -- match ANY|ALL
                    '(',      -- match (
                    'SELECT' -- match SELECT
                );
            if (foundPositions==nil) then
                found = false;
                break;
            end
            local operatorPos = foundPositions[1];
            local anyAllPos = foundPositions[2];
            local openBracketPos = foundPositions[3];
            searchStart = anyAllPos + 1
        
```

```

foundPositions = sqlparsing.find(tokens,
                                  openBracketPos,
                                  true,
                                  true,
                                  sqlparsing.iswhitespaceorcomment,
                                  ')');

if (foundPositions ~= nil) then
  local closeBracketPos = foundPositions[1]
  local operatorToken = tokens[operatorPos];
  local anyOrAll = sqlparsing.normalize(tokens[anyAllPos]);
  if (operatorToken=='<' or operatorToken=='<='
      or operatorToken==',' or operatorToken=='>=')
    then
      -- now we have <|<=|>|= ANY|ALL (SELECT <something> FROM
      -- rebuild to <|<=|>|= (SELECT MIN|MAX(<something>) FROM
      local setfunction = 'MIN';
      if ((anyOrAll=='ANY' or anyOrAll=='SOME') and
          (operatorToken=='<' or operatorToken=='<=')
        ) or
        (anyOrAll=='ALL' and (operatorToken==','
                               or operatorToken=='>=')
        )
      ) then
        setfunction = 'MAX';
    end
  tokens[anyAllPos] = '';
  tokens[openBracketPos] =
    '(SELECT ' .. setfunction .. '(anytab.anycol) FROM ()';
  tokens[closeBracketPos] = ')' as anytab(anycol))';
elseif (operatorToken=='=' and anyOrAll=='ALL') then
  -- special rebuild for = ALL
  -- rebuild to=(SELECT CASE WHEN COUNT(DISTINCT <something>)==1
  --                   THEN FIRST_VALUE(<something>) ELSE NULL END FROM
  tokens[anyAllPos] = '';
  tokens[openBracketPos] =
    '(SELECT CASE WHEN COUNT(DISTINCT anytab.anycol) = 1 \
               THEN FIRST_VALUE(anytab.anycol) ELSE NULL END FROM ()';
  tokens[closeBracketPos] = ')' as anytab(anycol))';
elseif ((operatorToken=='!= ' or operatorToken=='<> ')
        and anyOrAll=='ALL') then
  -- special rebuild for != ALL
  -- rebuild to NOT IN
  tokens[operatorPos] = ' NOT IN '
  tokens[anyAllPos] = ''
elseif (operatorToken=='!= ' and
       (anyOrAll=='ANY' or anyOrAll=='SOME')) then
  --special rebuild for != ANY, rebuild to
  -- CASE WHEN (SELECT COUNT(DISTINCT <something>) FROM ...) == 1
  -- THEN operand != (SELECT FIRST_VALUE(<something>) FROM ...)
  -- ELSE operand IS NOT NULL END
  --note: This case would normally require to determine the operand
  --      which requires full understanding of a value expression
  --      in SQL standard which is nearly impossible in
  --      preprocessing (and very susceptible to errors)
  --      so we evaluate the
  --      SELECT COUNT(DISTINCT <something>) FROM ... == 1 here and
  --      insert the correct expression
  --

```

```

-- first preprocess the inner query
local queryText = table.concat(tokens,
                                '',
                                openBracketPos,
                                closeBracketPos)
queryText = rebuildAnyAll( queryText )
-- since the subquery was already processed we can continue
-- searching *after* the SELECT
searchStart = closeBracketPos + 1
local distinctQueryText='SELECT COUNT(DISTINCT anytab.anycol) \
                           FROM '..queryText..' AS anytab(anycol)'
local success, result = pquery(distinctQueryText)
if (success) then
    if (result[1][1] == 1) then
        tokens[anyAllPos] = '(SELECT FIRST_VALUE(anystab.anycol) \
                               FROM '..queryText..' AS anytab(anycol))'
    else
        tokens[operatorPos] = ' IS NOT NULL '
        tokens[anyAllPos] = ''
    end
    -- clear all tokens of the SELECT
    for curTokenNr=openBracketPos,closeBracketPos do
        tokens[curTokenNr] = ''
    end
end
until found == false;
return table.concat(tokens);
end
/
CREATE SCRIPT sql_preprocessing.preprocessAnyAll AS
    import('sql_preprocessing.transformAnyAll', 'anyallparser');
    sqlparsing.setsqltext(
        anyallparser.rebuildAnyAll(sqlparsing.getsqltext()))
/
CREATE TABLE t1 (i INT);
INSERT INTO t1 VALUES 1,2,3,4,5,6,7,8,9,10;
CREATE TABLE t2 (j INT);
INSERT INTO t2 VALUES 5,6,7;

SELECT i FROM t1 WHERE i < ALL(SELECT j FROM t2);
Error: [0A000] Feature not supported: comparison with quantifier ALL

ALTER SESSION SET sql_preprocessor_script=
                sql_preprocessing.preprocessAnyAll;
SELECT i FROM t1 WHERE i < ALL(SELECT j FROM t2);

I
-----
1
2
3
4

```

3.9. Profiling

3.9.1. What is Profiling?

Exasol deliberately omits complex tuning mechanisms for customers, like e.g. execution hints, creation of different index types, calculation of table statistics etc. Queries in Exasol are analyzed by the query optimizer and corresponding tuning actions are executed fully automatically.

Nevertheless there exist situations where you want to know how much time the certain execution parts of a query take. Long running queries can then be analyzed and maybe rewritten. Furthermore this kind of information can be provided to Exasol to improve the query optimizer continuously.

In Exasol, you can switch on the profiling feature on demand. Afterwards, the corresponding information is gathered and provided to the customer via system tables. Further details are described in the following sections.

3.9.2. Activation and Analyzing

In default case, the profiling is only activated for the running queries and provides the current status through the system tables [EXA_DB_A_PROFILE_RUNNING](#) and [EXA_USER_PROFILE_RUNNING](#).

You can switch on the general profiling feature through the statements [ALTER SESSION](#) or [ALTER SYSTEM](#) by setting the option PROFILE. Afterwards, the corresponding profiling data is gathered during the execution of queries and is collected in the system tables [EXA_USER_PROFILE_LAST_DAY](#) and [EXA_DB_A_PROFILE_LAST_DAY](#). Please mention this profiling data is part of the statistical system tables which are committed just periodically. Therefore a certain delay occurs until the data is provided within the system tables. If you want to analyze the profiling information directly after executing a query, you can use the command [FLUSH STATISTICS](#) to enforce the COMMIT of the statistical data.

The profiling system tables list for each statement of a session certain information about the corresponding execution parts, e.g. the execution time (DURATION), the used CPU time (CPU), the memory usage (MEM_PEAK and TEMP_DB_RAM_PEAK) or the network communication (NET). Additionally, the number of processed rows (OBJECT_ROWS), the number of resulting rows (OUT_ROWS) and further information about a part (PART_INFO) is gathered. Please mention that some parts do not contain the full set of data.

The following execution parts exist:

COMPILE / EXECUTE	Compilation and execution of the statement (including query optimizing and e.g. the automatic creation of table statistics)
SCAN	Scan of a table
JOIN	Join with a table
FULL JOIN	Full outer join to a table
OUTER JOIN	Outer Join to a table
EXISTS	EXISTS computation
GROUP BY	Calculation of the GROUP BY aggregation
GROUPING SETS	Calculation of the GROUPING SETS aggregation
SORT	Sorting the data (ORDER BY, also in case of analytical functions)
ANALYTICAL FUNCTION	Computation of analytical functions (without sorting of data)
CONNECT BY	Computation of hierarchical queries
PREFERENCE PROCESSING	Processing of Skyline queries (for details see Section 3.10, “Skyline”)
PUSHDOWN	Pushdown SQL statement generated by the adapter for queries on virtual objects
QUERY CACHE RESULT	Accessing the Query Cache
CREATE UNION	Under certain circumstances, the optimizer can create a combined table out of several tables connected by UNION ALL, and process it much faster
UNION TABLE	This part is created for each individual UNION ALL optimized table (see also part "CREATE UNION")
INSERT	Inserting data (INSERT, MERGE or IMPORT)

UPDATE	Update of data (UPDATE or MERGE)
DELETE	Deleting data (DELETE, TRUNCATE or MERGE)
IMPORT	Execution of the IMPORT command
EXPORT	Execution of the EXPORT command
COMMIT	Commit of the transaction, persistent write to hard disk
ROLLBACK	Rollback of the transaction
WAIT FOR COMMIT	Waiting until another transaction finishes
INDEX CREATE	Creation of internal indices
INDEX INSERT	Insert in internal indices
INDEX UPDATE	Update on internal indices
INDEX REBUILD	Recreation of internal indices
CONSTRAINT CHECK	Checking constraints (primary/foreign key, NOT NULL)
DISTRIBUTE	Distribution across the cluster nodes
RECOMPRESS	Recompressing data
REPLICATE	Cluster-wide replication of data (e.g. replicating small tables to avoid global joins)
SYSTEM TABLE SNAPSHOT	Collecting the data for a system table

3.9.3. Example

The following example shows how the profiling data looks like for a simple query:

```
-- omit autocommits and switch on profiling
SET AUTOCOMMIT OFF;
ALTER SESSION SET profile='ON';

-- run query
SELECT YEAR(o_orderdate) AS "YEAR", COUNT(*)
FROM orders
GROUP BY YEAR(o_orderdate)
ORDER BY 1 LIMIT 5;

YEAR  COUNT(*)
-----
1992      227089
1993      226645
1994      227597
1995      228637
1996      228626

-- switch off profiling again and avoid transaction conflicts
ALTER SESSION SET profile='OFF';
COMMIT;
-- provide newest information and open new transaction
FLUSH STATISTICS;
COMMIT;

-- read profiling information
SELECT part_id, part_name, object_name, object_rows, out_rows, duration
FROM exa_user_profile_last_day
WHERE CURRENT_STATEMENT=5 = stmt_id AND CURRENT_SESSION=session_id;

PART_ID PART_NAME          OBJECT_NAME        OBJECT_ROWS OUT_ROWS DURATION
-----
1  COMPILE / EXECUTE           ORDERS        1500000  1500000  0.020672
2  SCAN                         ORDERS        1500000  1500000  0.247681
```

3 GROUP BY	tmp_subselect0	0	7	0.020913
4 SORT	tmp_subselect0	7	5	0.006341

3.10. Skyline

3.10.1. Motivation

When optimal results for a specific question shall be found for big data volumes and many dimensions, certain severe problems arise:

1. Data flooding

Large data volumes can hardly be analyzed. When navigating through millions of data rows, you normally sort by one dimension and regard the top N results. By that, you will nearly always miss the optimum.

2. Empty result

Filters are often used to simplify the problem of large data volumes. Mostly those are however too restrictive which leads to completely empty result sets. By iteratively adjusting the filters, people try to extract a controllable data set. This procedure generally prevents finding the optimum, too.

3. Difficulty of correlating many dimensions

When many dimensions are relevant, you can hardly find an optimum via normal SQL. By the use of metrics, analysts try to find an adequate heuristic to weight the different attributes. But by simplifying the problem to only one single number, a lot of information and correlation within the data is eliminated. The optimum is also mostly missed by this strategy.

In summary, analysts often navigate iteratively through large data volumes by using diverse filters, aggregations and metrics. This approach is time consuming and results in hardly comprehensible results, whose relevance often keeps being disputable. Instead of a real optimum for a certain question, only a kind of compromise is found which disregards the complexity between the dimensions.

3.10.2. How Skyline works

Skyline is Exasol's self-developed SQL extension which solves the problems of data analysis described above. With Skyline, users get an intuitive tool for finding the optimal result for a large number of dimensions (multi-criteria optimization).

Instead of hard filters via the WHERE clause and metrics between the columns, the relevant dimensions are simply specified in the PREFERRING clause. Exasol will then determine the actual optimum. The optimal set means the number of non-dominated points in the search space, also named as Pareto set. By definition, a point is dominated by another one if it is inferior in all dimensions.

For better illustration, please imagine an optimization space with just two dimensions, for example the decision for a car by using the two attributes "high power" and "low price". A car A is consequently dominated by a car B if its price is lower and its performance is higher. If only one dimension is superior, but the other one is inferior, then you cannot find an obvious decision.

In consequence, the Pareto set means in this case the set of cars with preferably high performance and low price. Without Skyline, you can only try to find a reasonable metric by combining the performance and price in a formula, or by limiting the results by certain price or performance ranges. But the actual optimal cars won't be identified by that approach.

With Skyline, you can simply specify the two dimensions within the PREFERRING clause (PREFERRING HIGH power PLUS LOW price) and Exasol will return the optimal combinations. You should bring to mind that the Skyline algorithm compares all rows of a table with all others. In contrast, by using a simple metric the result can be easily determined (sorting by a number), but the complexity of the correlations is totally eliminated. Skyline hence gives you the possibility to actually consider the structure of the data and to find the real optimal result set.

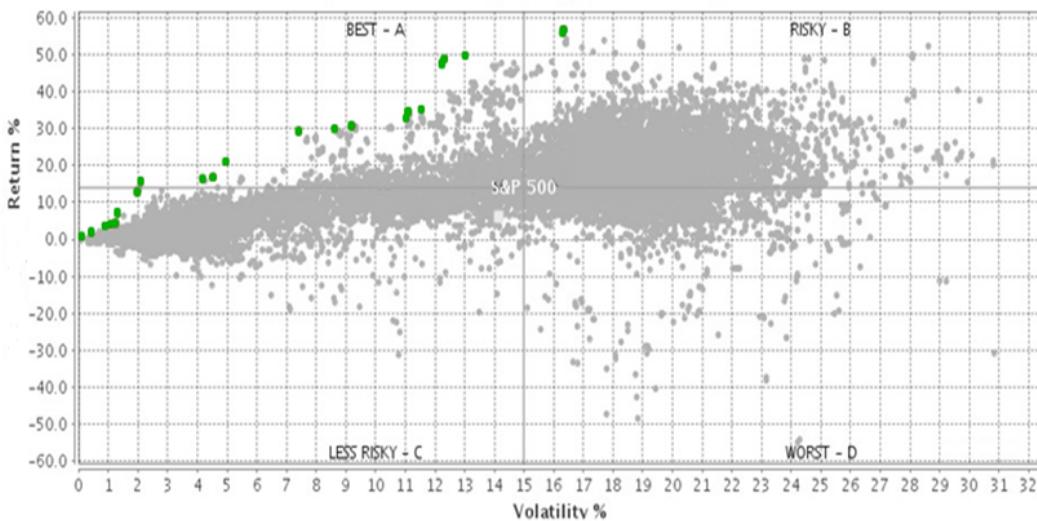
The advantages of Skyline could hopefully be demonstrated by the simple two-dimensional example. But the algorithm can also consider a large number of dimensions which the human brain can hardly imagine any more.

The PREFERRING clause can by the way be used in the **SELECT**, **DELETE** and **UPDATE** statements.

3.10.3. Example

To illustrate the power of Skyline, we consider the selection of the best funds in the market which is a daily problem for financial investors. To select good funds, one can use a lot of attributes, for example its performance, volatility, investment fees, ratings, yearly costs, and many more.

To simplify the problem, we want to concentrate on the first two attributes. In reality, the performance and volatility have a reversed correlation. The more conservative a fund is, typically the lower its volatility, but also its performance.



```
SELECT * FROM funds PREFERRING HIGH performance PLUS LOW volatility;
```

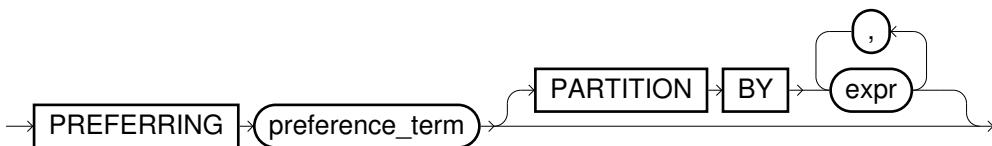
In the picture above, thousands of funds are plotted by its performance and volatility. The green points are the result of given Skyline query. These funds represent the optimal subset regarding the two dimensions. Afterwards, the decision whether one selects a more conservative or more risky fund can be done in a subsequent, subjective step.

But you can already see in this pretty simple example how much the problem can be reduced. Out of thousands of funds, only the best 23 are extracted, and this subset is the actual optimum for the given problem.

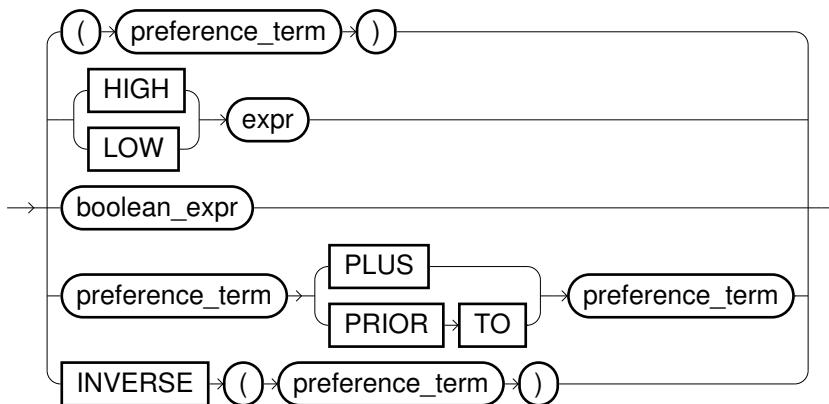
3.10.4. Syntax elements

As shown in the description of the statement **SELECT**, the PREFERRING clause can be defined after the WHERE condition and can contain the following elements:

`preferring_clause ::=`



preference_term::=



PARTITION BY

If you specify this option, then the preferences are evaluated separately for each partition.

HIGH and LOW

Defines whether an expression should have high or low values. Please note that numerical expressions are expected here.

Boolean expressions

In case of boolean expressions, the elements are preferred where the condition results in TRUE. The expression `x>0` is therefore equivalent to `HIGH (x>0)`. The latter expression would be implicitly converted into the numbers 0 and 1.

PLUS

Via the keyword PLUS, multiple expressions of the same importance can be specified.

PRIOR TO

With this clause you can nest two expressions hierarchically. The second term will only be considered if two elements have the similar value for the first term.

INVERSE

By using the keyword INVERSE, you can create the opposite/inverse preference expression. Hence, the expression `LOW price` is equivalent to `INVERSE(HIGH price)`.

The following, more complex example shows the selection of the best cars with nested expressions:

- The performance is segmented by steps of tens.
- The price is segmented by steps of thousands.
- In case of identical performance and price, the color "silver" is preferred.

```

SELECT * FROM cars
  PREFERRING (LOW ROUND(price/1000) PLUS HIGH ROUND(power/10))
    PRIOR TO (color = 'silver');
  
```


Chapter 4. Clients and interfaces



All clients and interfaces can be downloaded using the Exasol customer portal (wwwexasol.com [<http://wwwexasol.com/>]).

4.1. EXAplus

EXAplus is an user interface for dealing with [SQL](#) statements in Exasol. It is implemented in Java and is available as graphical application and simple console version both under Windows and Linux.

4.1.1. Installation

MS Windows

EXAplus has been successfully tested on the following systems:

- Windows 10 (x86/x64)
- Windows 7, Service Pack 1 (x86/x64)
- Windows Server 2012 R2 (x86/x64)
- Windows Server 2012 (x86/x64)
- Windows Server 2008 R2, Service Pack 1 (x86/x64)
- Windows Server 2008, Service Pack 2 (x86/x64)

To install EXAplus, please follow the instructions of the installation wizard which will be started when executing the installation file.



Administrator rights and the Microsoft .NET Framework 4.0 Client Profile TMare required for installing EXAplus.

In addition you need to consider the following requirements:

- EXAplus requires a Java runtime environment for execution. To be able to use all features of EXAplus we recommend to use at least Java 7 and applied updates.

Further it is recommended that *support for additional languages* was selected during the Java installation. For correct formatting of special unicode characters (e.g. Japanese characters) a font that is capable of displaying those characters must be installed on your Windows system.

- After you have successfully installed EXAplus, a corresponding start menu entry is added to start the graphical EXAplus version. But you can also use EXAplus in your commandline interface (cmd.exe) by calling `exaplus64.exe` or `exaplus.exe` (32-bit executable). If you do not want to type in the complete path to EXAplus, please choose the option *Add EXAplus to path* during the installation.

Linux/Unix

EXAplus has been successfully tested on the following systems:

- Red Hat / CentOS 7, OpenJDK JVM 1.8.0 (x64)
- Red Hat / CentOS 6, OpenJDK JVM 1.8.0 (x86/x64)
- Debian 8, OpenJDK JVM 1.7.0 (x86/x64)
- Ubuntu 16.04 LTS, OpenJDK JVM 1.8.0 (x86/64)
- Ubuntu 14.04 LTS, OpenJDK JVM 1.7.0 (x86/64)
- SUSE Linux Enterprise Server 12, IBM's JVM 1.7.0 (x64)

- SUSE Linux Enterprise Desktop 12, OpenJDK JVM 1.7.0 (x64)
- SUSE Linux Enterprise Server 11 Service Pack 3, IBM's JVM 1.7.0 (x86/x64)
- openSUSE Leap 42.2, OpenJDK JVM 1.8.0 (x64)
- macOS Sierra, JVM 1.8.0 (64Bit)
- OS X 10.11, JVM 1.8.0 (64Bit)
- FreeBSD 11.0, OpenJDK 1.8.0 (64Bit)
- FreeBSD 10.3, OpenJDK 1.8.0 (64Bit)

In addition you need to consider the following requirements:

- Java 7 or higher has to be installed and the program "java" must be included in the path.

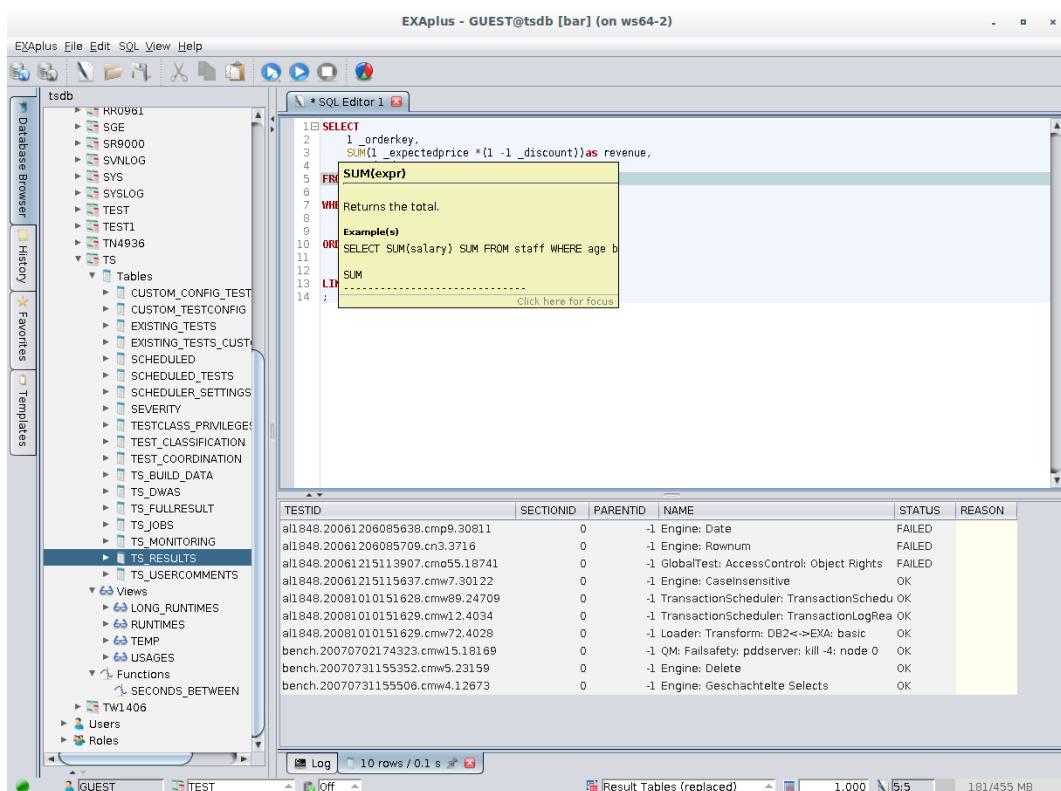
For correct formatting of special unicode characters (e.g. Japanese characters), a font that is capable of displaying those characters must be installed on your system.

- If you want to use Japanese characters and your Input Manager is Scim, we recommend to install at least version 1.4.7 of Scim. Additionally, Scim-Anthy (Japanese character support) should be installed in version 1.2.4 or higher.

The EXAplus installation archive can be unpacked with the command **tar -xfz**. If the above software requirements are met, EXAplus can now be started by invoking the startup script **exaplusgui** (graphical version) or **exaplus** (commandline version). A separate installation is not necessary.

4.1.2. The graphical user interface

The graphical user interface of EXAplus looks like as follows:



The following areas can be distinguished:

Menu bar

Execution of different actions via the menu.

Toolbar	Direct execution of common actions via icons.
Database Browser	Shows the existing database objects like schemas, tables, scripts, users, etc. When doing a double click on the object an object information will be opened in the Editor area. In the context menu of the objects (right-click) you will find a list of possible actions you can execute on this object.
History	Overview over the recently executed commands. If you want to save a query as favorite, you can do this via the context menu in the History (right-click on the SQL).
Favorites	Preferred or frequently used queries can be saved as a favorite. You can import and export your favorites via the context menu (right-click on a folder).
Templates	Lists a bunch of templates for SQL statements and functions, Lua functions and EXAplus commands. This is a quick reference if you don't remember the syntax details or e.g. the list of available string functions. Via Drag&Drop you can copy the templates into your SQL editor.
SQL Editor	Working area to display, edit and execute SQL scripts. Moreover, the object information is opened here.
Result Area	Display of query results in table form and the log protocol of EXAplus.
Status Bar	Shows the current connection information and configuration which can be directly changed in part (e.g. the Autocommit mode or the limitation of result tables). You can also see the current memory allocation.

Most features of EXAplus will be intuitive, but in the following table you may find some topics which facilitate the daily work.

Table 4.1. Information about the work with EXAplus

Topic	Annotation						
Connections	<p>When starting EXAplus, a dialog is opened where you can define your connection data. This data is stored by EXAplus for the next start and can be reused easily (except the password). Alternatively, this dialog is opened when clicking on the connect icon in the toolbar.</p> <p>If a user uses several databases or wants to connect to a database through multiple user profiles, the usage of connection profiles will be useful. Those can be created and configured through the menu entry <i>EXAplus -> Connection Profiles</i>.</p> <p>Afterwards you can choose these profiles when clicking on the connect icon or via the menu entry <i>EXAplus -> Connect To</i>. If some parts like the password was not already configured in the profile, you have to complete them in the connection dialog. After profiles are created, the Custom connection dialog doesn't appear any more when starting EXAplus.</p> <p>To automate the connection process completely, you can define a default connection which will then be used when EXAplus is started.</p> <p> You can also use the created connections in Console mode (commandline parameter <i>-profile</i>)</p>						
Multiple EXAplus instances	<p>If a user wants to connect to multiple databases or wants to open multiple sessions to a single one, he can start multiple instances of EXAplus by starting EXAplus multiple times or via the menu entry <i>EXAplus -> New Window</i>.</p> <p> Per instance, EXAplus allows only one open connection to the database.</p> <p>Preference changes are applied by all open EXAplus instances.</p>						
Result tables	<p>Via status bar or menu you can specify whether old result tables are retained or deleted before executing a new SQL command or whether result tables shall be displayed in text format (Log).</p> <p>Moreover, you can define the maximal amount of rows which is displayed. This option limits the data volume which is sent from the database to the client and can be useful especially in case of big result sets.</p> <p>Result tables can be reordered by clicking on a certain column. The content of result tables can be copied to external applications like Excel via Copy & Paste.</p>						
Current schema	<p>In the status bar you can find and switch the current schema. Additionally, you can restrict the list of displayed schemas in the database browser via the menu entry <i>View -> Show Only Current Schema</i>.</p>						
Autocommit mode	<p>The autocommit mode is displayed in the status bar and can also be switched there. The following values are possible:</p> <table> <tr> <td>ON</td> <td>An implicit COMMIT is executed after each SQL statement (default).</td> </tr> <tr> <td>OFF</td> <td>An implicit COMMIT is never executed.</td> </tr> <tr> <td>EXIT</td> <td>An implicit COMMIT is executed only when closing the connection.</td> </tr> </table>	ON	An implicit COMMIT is executed after each SQL statement (default).	OFF	An implicit COMMIT is never executed.	EXIT	An implicit COMMIT is executed only when closing the connection.
ON	An implicit COMMIT is executed after each SQL statement (default).						
OFF	An implicit COMMIT is never executed.						
EXIT	An implicit COMMIT is executed only when closing the connection.						
Drag&Drop	<p>Via Drag&Drop you can copy SQL commands from the history directly in the editor area.</p> <p>Equally you can insert the schema-qualified name of a database object (e.g. MY_SCHEMA.MY_TABLE) from the database browser into the editor area. Only in case of columns the schema name is omitted.</p>						

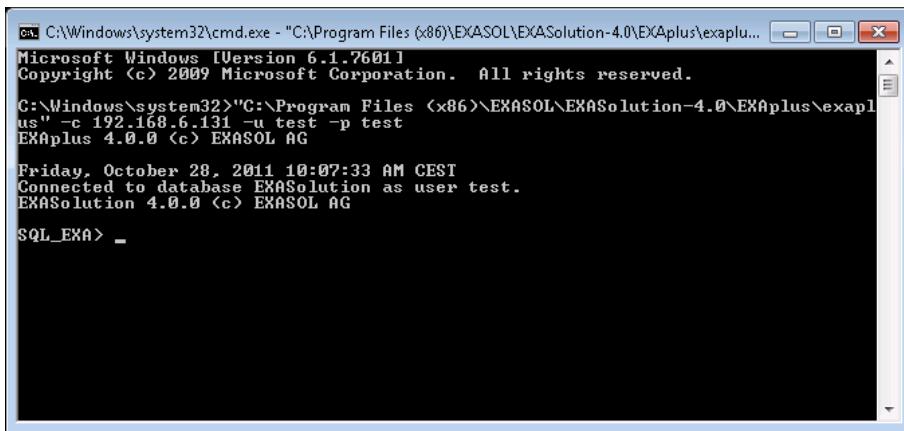
Topic	Annotation
Autocompletion	<p>By executing the shortcut <i>CTRL+SPACE</i> EXAplus tries to complete the current word in the editor by considering metadata of the database (like column names of tables or table names in the current scope). Additionally, information and examples are provided for e.g. functions.</p> <p>Furthermore, via the shortcut <i>SHIFT+CTRL+SPACE</i>, EXAplus tries to autocomplete the current word in a simple way by taking the match of the least recently typed words. If you use the same name (e.g. a schema name) within a query, then this feature can simplify your work.</p>
Editor Mode	<p>Via the context menu (right-click on any position within the SQL editor) you can switch between the different editor modes (SQL, LUA, R, PYTHON). This influences the features autocompletion, syntax highlighting and code folding.</p>
Bookmarks	<p>You can create bookmarks by clicking in the SQL editor on the gray border on the left beside the line numbers or by using the shortcut <i>CTRL+F2</i>. Subsequently, you can jump between the next bookmark via the keys <i>F2</i> and <i>SHIFT+F2</i>.</p>
Parametrized SQL scripts	<p>By the use of variables you can parametrize SQL scripts. For those variables you can even request dynamical user input. Details can be found in Section 4.1.4, “EXAplus-specific commands”.</p>
Object info	<p>By doing a double click on objects or via the context menu entry <i>Info</i> you can open an object information tab in the editor area including several information about the object: Examples are:</p> <ul style="list-style-type: none"> • Schemas (including the list of the contained schema objects) • Tables (including the list of columns and constraints) • Views and scripts (including the creation text) • User and roles (including the list of system privileges) • Result tables (including information about the execution) • SQL commands from the history (including information about the execution) <p>On the top of the object info you find some useful icons to e.g. automatically create the corresponding DDL statements of this object (also available for whole schemas), drop the object, export a table or view or edit views or scripts.</p> <p>Those and more actions can also be found in the context menu of a database browser object.</p>
System monitoring	<p>By clicking the pie chart button on the tool bar, you can open a separate window showing graphical usage statistics of the connected database. You can add more statistics, let graphs stack upon each other or be displayed in a grid, and select the interval, type, and data group for each graph. The graphs are updated about every 3 minutes.</p>
Language settings	<p>The preferable language can be switched in the <i>EXAplus -> Preferences</i>.</p>
Error behavior	<p>Via the menu entry <i>SQL -> Error Handling</i> you can specify the behavior in case of an error. See also WHENEVER command for details.</p>
Memory Usage	<p>The bar at the bottom-right corner shows the memory allocation pool (heap) usage of the current EXAplus process. By a double-click you can hint the program to trigger a garbage collection.</p>
Associate EXAplus with SQL files	<p>If you switch on this option during the installation, you can open SQL files in EXAplus by a double-click (on Windows systems). For encoding the last preferences are used when a file was opened or saved.</p>

Topic	Annotation
Migration of EXAplus preferences	<p>If you want to migrate the preferences e.g. to another computer, you can easily copy the corresponding xml files from the EXAplus folder (exasol or .exasol) in your home directory. Here you will find the Favorites (favorites.xml), the History (history*.xml), the connection profiles (profiles.xml) and the general EXAplus preferences (exaplus.xml).</p> <p> The xml files should only be copied for an EXAplus instance with similar version.</p>

 In the background, EXAplus opens a second connection to the database which requests meta data for e.g. the Database Browser or the autocompletion feature. Please mention that therefore database changes cannot be displayed in the Database Browser unless they are not committed in the main connection.

4.1.3. The Console mode

Alongside the graphical version EXAplus is also applicable in plain console mode which is very useful especially for batch jobs. You can start the console version by invoking **exaplus64[.exe]** or **exaplus[.exe]** (32-bit executable).



The screenshot shows a Windows command prompt window titled 'cmd C:\Windows\system32\cmd.exe - "C:\Program Files (x86)\EXASOL\EXASolution-4.0\EXAplus\exaplus"'. The window displays the following text:

```

Microsoft Windows [Version 6.1.7601]
Copyright <c> 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>"C:\Program Files (x86)\EXASOL\EXASolution-4.0\EXAplus\exaplus" -c 192.168.6.131 -u test -p test
EXAplus 4.0.0 <c> EXASOL AG

Friday, October 28, 2011 10:07:33 AM CEST
Connected to database EXASolution as user test.
EXASolution 4.0.0 <c> EXASOL AG

SQL_EXA> _

```

After successfully connecting to the database an interactive session is started where the user can send SQL commands (see [Chapter 2, *SQL reference*](#)) or EXAplus commands (see [Section 4.1.4, “EXAplus-specific commands”](#)) to the database. Result tables are then displayed in text form.

Notes:

- You can exit EXAplus via the commands **EXIT** and **QUIT**
- All commands have to be finished by a semicolon (except view, function and script definitions which must be finished by a / in a new line and except the commands **EXIT** and **QUIT**)
- The command history can be used via the arrow keys (up/down)
- Multiline commands can be aborted by using the shortcut **<CTRL>-T**
- By pressing **<TAB>** EXAplus tries to complete the input

Table 4.2. EXAplus command line parameters (Console mode only)

Parameter	Annotation
Help options	
<code>-help</code>	Displays a short overview about the command line parameters.
<code>-version</code>	Displays version information.
Connection options	
<code>-c <connection string></code>	Connection data which consists of the cluster nodes and the port (e.g. 192.168.6.11..14:8563).
<code>-u <user></code>	User name for logging-in to Exasol.
<code>-p <passwd></code>	Password for logging-in to Exasol.
<code>-s <schema></code>	EXAplus will attempt to open the specified schema after the log-in to Exasol.
<code>-L</code>	Only one login attempt is performed. If the user name and password are specified in the command line, no enquiry is made in the event of an incorrect password. This parameter is very useful for utilization in scripts.
<code>-encryption <ON/OFF></code>	Enables the encryption of the client/server communication. Default: ON
<code>-profile <profile name></code>	Name of connection profile defined in <code><configDir>/profiles.xml</code> (changeable via EXAplus GUI or with profile handling parameters).
<code>-k</code>	Use Kerberos based single sign-on.
Profile handling options	
<code>-lp</code>	Print a list of existing profiles and exit.
<code>-dp <profile name></code>	Delete a specified profile and exit.
<code>-wp <profile name></code>	Write a specified profile and exit. The profile is defined by connection options.
File options	
SQL files can be integrated via the parameters <code>-f</code> or <code>-B</code> or via the EXAplus commands <code>start</code> , <code>@</code> and <code>@@</code> (see also Section 4.1.4, “EXAplus-specific commands”). EXAplus will search the specified files firstly relatively to the working directory, but also in the folders which are defined in the environment variable <code>SQLPATH</code> . If the files are not found, EXAplus also tries to add an implicit <code>.sql</code> and <code>.SQL</code> ending. The variable <code>SQLPATH</code> contains similar to the variable <code>PATH</code> a list of folder, separated by an “ <code>:</code> ” under Linux and by an “ <code>;</code> ” under Windows.	
<code>-init <file></code>	File which is initially executed after the start. Default: <code>exaplus.sql</code>
<code>-f <file></code>	EXAplus executes that script and terminates.
<code>-B <file></code>	EXAplus executes that script in batch mode and terminates.
<code>-encoding <encoding></code>	Sets the character set for reading of SQL-scripts started with <code>-f</code> or <code>-B</code> . For supported encodings see Appendix D, Supported Encodings for ETL processes and EXAplus . As default the character set UTF8 is used. By using the command <code>SET ENCODING</code> you can also change the character set during a session. But you have to consider that the change has no impact on already opened files.

Parameter	Annotation				
-- <args>	<p>SQL files can use arguments given over via the parameter “--” by evaluating the variables &1, &2 etc..</p> <p>For example, the file <code>test.sql</code> including the content</p> <pre>--test.sql SELECT * FROM &1;</pre> <p>can be called in the following way:</p> <p><code>exaplus -f test.sql -- dual</code></p>				
Reconnect options					
-recoverConnection <ON/OFF>	<p>Attempts to recover a lost connection.</p> <p>Default: OFF</p>				
-retry <num>	<p>Determines the number of attempts to recover a connection. If either -1 or UNLIMITED is specified, repeated attempts will be made to recover the connection indefinitely.</p> <p>Default: UNLIMITED</p>				
-retryDelay <num>	<p>Minimum interval between two recovery attempts in seconds.</p> <p>Default: 5</p>				
-closeOnConnectionLost	<p>Exits EXAplus after the loss of a connection that cannot be recovered.</p> <p>Default: ON</p>				
Other options					
-autocommit <ON/OFF/EXIT>	<p>Sets the autocommit mode. ON enables autocommit, OFF disables it. EXIT effects an autocommit when the program is exited or disconnected.</p> <p>Default: ON</p>				
-lang <EN/DE/JA>	<p>Defines the language of EXAplus messages.</p> <p>Default: Depends on the system preferences.</p>				
-characterwidth <HALF/FULL>	<p>Defines the width of unicode characters when displaying a query result.</p> <p>Default:</p> <table style="margin-left: 40px;"> <tr> <td>lang=EN or DE</td> <td>HALF</td> </tr> <tr> <td>lang=JA</td> <td>FULL</td> </tr> </table>	lang=EN or DE	HALF	lang=JA	FULL
lang=EN or DE	HALF				
lang=JA	FULL				
-q	Quiet mode which suppresses additional output from EXAplus.				
-x	Stops EXAplus in the event of errors.				
-F <num>	<p>Determines the fetchsize in kB which specifies the amount of resultset data is sent during one single communication step with the database.</p> <p>Default: 2000</p>				
-Q <num>	<p>Query Timeout in seconds. A query will be aborted if the timeout is exceeded.</p> <p>Default: -1 (unlimited)</p>				

Parameter	Annotation
<code>-autoCompletion <ON/OFF></code>	If this function is enabled, the user can obtain proposals by pressing TAB. When using scripts, this prediction aid will be automatically deactivated. Default: ON
<code>-pipe</code>	With this parameter you can use pipes on Linux/Unix systems (e.g. echo "SELECT * FROM dual;" exaplus -pipe -c ... -u ... -p ...). Default: OFF
<code>-sql <SQL statement></code>	By this parameter you can execute single SQL statements. EXAplus quits afterwards. Example: <code>-sql "SELECT \"DUMMY\" FROM dual;"</code>
<code>-jdbcp param <JDBC parameter></code>	Set additional JDBC parameters.

4.1.4. EXAplus-specific commands

Overview

Command	Function
File and operating system commands	
<code>@</code> and <code>START</code>	Loads a text file and executes the statements contained therein.
<code>@@</code>	Loads a text file and executes the statements contained therein. However, if this statement is used in a SQL script, the search path begins in the folder in which the SQL script is located.
<code>HOST</code>	Performs an operating system command and then returns to EXAplus.
<code>SET SPOOL ROW SEPARATOR</code>	Defines the row separator for the <code>SPOOL</code> command.
<code>SPOOL</code>	Saves the input and output in EXAplus to a file.
Controlling EXAplus	
<code>BATCH</code>	Switches batch mode on or off.
<code>CONNECT</code>	Establishes a new connection with Exasol.
<code>DISCONNECT</code>	Disconnects the current connection with the database.
<code>EXIT</code> and <code>QUIT</code>	Terminates all open connections and closes EXAplus.
<code>PAUSE</code>	Issues some text in the console and waits for confirmation.
<code>PROMPT</code>	Issues some text in the console.
<code>SET AUTOCOMMIT</code>	Controls whether Exasol should automatically perform COMMIT statements.
<code>SET AUTOCOMPLETION</code>	Switches auto-completion on or off.
<code>SHOW</code>	Displays EXAplus settings.
<code>TIMING</code>	Controls the built-in timer.
<code>WHENEVER</code>	Defines the behavior of EXAplus in the event of errors.
Formatting	
<code>COLUMN</code>	Shows the formatting settings or amends them.
<code>SET COLSEPARATOR</code>	Sets the string that separates two columns
<code>SET ENCODING</code>	Selects a character set or outputs the current one.
<code>SET ESCAPE</code>	Sets the escape character, which makes it possible to input special characters.

Command	Function
SET FEEDBACK	Controls the output of confirmations.
SET HEADING	Switches the output of column headings on or off.
SET LINESIZE	Sets width of the output lines.
SET NULL	Defines the string for displaying NULL values in tables.
SET NUMFORMAT	Sets the formatting of numeric columns in tables.
SET PAGESIZE	Sets how many lines of text there should be before column headings are repeated.
SET TIME	Switches output of the current time of the client system at the input prompt on or off.
SET TIMING	Switches display of the time needed for execution of an SQL statement on or off.
SET TRUNCATE HEADING	Defines whether column headings are truncated or not.
SET VERBOSITY	Switches additional program information on or off.

Statements for handling variables

The user can define any number of variables, which remain effective for the duration of the EXAplus session or until explicitly deleted with an appropriate statement. The value of a variable can be accessed in all statements with &variable. Variables are always treated as strings.

ACCEPT	Receives input from the user.
DEFINE	Assigns a value to a variable.
SET DEFINE	Sets the characters with which the user variables are initiated.
UNDEFINE	Deletes a variable.

Alphabetical list of EXAplus commands

@ and START

Syntax

```
@ <file> [args];
```

Description

Loads a text file and executes the statements contained therein. @ and START are synonymous.

If no absolute path is specified, the file is searched for in relation to the working directory of EXAplus. Rather than local paths, it is also possible to use http and ftp URLs. If the file cannot be opened at the first attempt, the extension .sql is appended to the name and another search is made for the file.

A script can be given any number of arguments via command line parameter (Console mode), which can be called from the script as &1, &2 ... The variable &0 contains the name of the script. These variables are only defined during the lifetime of the script.

Example(s)

```
@test1.sql 2008 5;
@ftp://frank:swordfish@ftp.scripts/test2.sql;
@http://192.168.0.1/test3.sql;
```

@@

Syntax

```
@@ <file> [args];
```

Description

Similar to @ or START, however, the file is searched for in the path in which the called script is located (if the call is made through a script, otherwise the working directory of EXAplus is used).

Example(s)

```
--Contents of file /home/mb/test1.sql:  
@@test2;  
  
--File /home/mb/test2.sql is executed  
SQL_EXA> @/home/mb/test1.sql;
```

ACCEPT

Syntax

```
ACCEPT <variable> [PROMPT <text>] [DEFAULT <text>];
```

Description

Receives the value of the <variable> variable from the user as a keyboard input. If the *prompt* parameter is entered, the specified <text> text will be output beforehand. The value submitted via the <default> parameter is taken as the requirement if the user simply presses the return button.

Example(s)

```
SQL_EXA> accept ok prompt "Everything ok? " default "maybe";  
Everything ok? yes  
SQL_EXA>
```

BATCH

Syntax

```
BATCH BEGIN|END|CLEAR;
```

Description

Switches batch mode on or off. In this mode, SQL statements are not immediately executed but bundled, this can increase the execution speed.

BEGIN Switches to batch mode. From this point all entered SQL statements are saved and only executed after BATCH END has been entered.

CLEAR Deletes all stored statements.

END Runs all SQL statements entered since the last **BATCH BEGIN** and exits batch mode.

 If an exception is raised for a statements in the batch mode, the following statements are not executed anymore and the status before the batch execution is restored. A single exception is the **COMMIT** statement which finishes a transaction and stores changes persistently.

 If EXAplus is in batch mode, only specific statements are accepted:

- SQL statements (see [Chapter 2, SQL reference](#)) except scripts
- **DEFINE** and **UNDEFINE**
- **START**, **@** and **@@**
- **PAUSE**
- **PROMPT**
- **SHOW**
- **SPOOL**

Example(s)

```
BATCH BEGIN;
insert into t values(1,2,3);
insert into v values(4,5,6);
BATCH END;
```

COLUMN

Syntax

```
COLUMN [<name>]; -- Displays configuration
COLUMN <name> <command>; -- Changes configuration
```

Description

Displays the formatting settings for the **<name>** column or changes the formatting options for the **<name>** column. **<command>** can be a combination of one or more of the following options:

ON	Enables formatting settings for the relevant column.
OFF	Disables formatting settings for the relevant column.
FORMAT <format>	Sets the <format> format string for the values of the relevant column. Formatting options are dependent on the type of column. Therefore, format strings for text columns should not be used on numeric columns. Numeric values are right-aligned and other data types left-aligned. Format strings for alphanumeric columns have the form a<num> , where <num> indicates the number of letters. The width of a single column cannot be greater than the maximum length of a row.

Example: **a10** formats a column to a width of 10.

Format strings of numbers consist of the elements '**9**', '**0**', '**.**' (point) and '**EEEE**'. The width of the column results from the length of the specified format string.

9	At this position one digit is represented if it is not a NULL before or after the decimal point.
0	At this position one digit is always represented, even if it a NULL before or after the decimal point.

. (point)	Specifies the position of the point. The decimal point is represented as a point.
EEEE	The number is represented in scientific notation with exponents. Exactly four Es must be contained in the format string, because the width of the column is extended by four characters due to the exponents.

Apply the following rules:

- There must be at least one nine or zero.
- The decimal point is optional and it only may occur once.
- If there is a decimal point, a nine or zero must be both before and after it.
- Left to the decimal point nines are only allowed before zeros and right to it vice versa.
- The four E's are optional and may occur only once at the end in the format. If this scientific notation is chosen, numbers will be normalized to one digit to the left of the decimal point.
- All characters are case-insensitive.

Example: 99990 . 00 formats a number to the width of eight places (seven digits plus the decimal point). There will be displayed up to five but at least one digit to the left and exactly two digits to right of the decimal point. In case of the number zero for example 0 . 00 will be shown, in case of 987 . 6543 there will be printed 987 . 65 whereas it is rounded mathematically. A # will be displayed for each digit if a value is not presentable because it requires too many places left of the decimal point.

CLEAR	Deletes the formatting options for the relevant column.
JUSTIFY <LEFT RIGHT CENTER>	Defines the alignment of the column name.
LIKE <column>	Copies all formatting options from the specified column.
WORD_WWRAPPED	The values of the relevant column will, if possible, be wrapped between the individual words.
WRAPPED	The values of the relevant column will be wrapped at the right margin of the column.
TRUNCATED	The values of the relevant column will be trimmed at the right margin of the column.
NULL <text>	NULL values of the relevant column will be displayed as <text>.
HEADING <text>	Sets a new heading for the relevant column.
ALIAS <text>	Sets an alias name for the relevant column.

The following "COLUMN" SQL*Plus statements are not supported. However, for reasons of compatibility they will not generate a syntax error:

- NEWLINE
- NEW_VALUE
- NOPRINT
- OLD_VALUE
- PRINT
- FOLD_AFTER
- FOLD_BEFORE

Example(s)

```
SQL_EXA> column A;
COLUMN A ON
FORMAT 9990

SQL_EXA> column A format 90000.0;
```

```
SQL_EXA> column b format 99990;
SQL_EXA> column b just left;
SQL_EXA> select a,a as b from ty;
```

A	B
0011.0	11
0044.0	44
0045.0	45
0019.0	19
0087.0	87
0099.0	99
0125.0	125
0033.0	33
0442.0	442

```
0011.0 11
0044.0 44
0045.0 45
0019.0 19
0087.0 87
0099.0 99
0125.0 125
0033.0 33
0442.0 442
```

CONNECT

Syntax

```
CONNECT <user>[ /<password> ][@<connection string>];
```

Description

A new connection with Exasol can be established with this command.

If a connection already exists, it will be disconnected if the new connection was established successfully. If no password is specified, this is requested by EXAplus. If `<connection string>` is not specified, the information of the last connection is used. A COMMIT will also be performed if "SET AUTOCOMMIT EXIT" has been set.

Example(s)

```
CONN scott/tiger;
CONNECT scott/gondor@191.168.2.1:8563;
```

DEFINE

Syntax

```
DEFINE [<variable>[=<value>]];
```

Description

Assigns the string `<value>` to the variable `<variable>`. Single and double quotes have to be doubled like in SQL strings. If this variable does not exist, it will be created. If DEFINE is only called with the name of a variable, the value of the variable will be displayed. As a result of calling DEFINE without parameters, all variables and the assigned values will be listed.

 The dot is used as delimiter for variable names (e.g. after calling `define v=t` the string `&v.c1` is replaced by `t.c1`). That's why if you want to add a dot you have to specify two dots (`&v..c1` will be evaluated as `t.c1`)

Example(s)

```
define tablename=usernames;
define message='Action successfully completed.';
```

DISCONNECT

Syntax

DISCONNECT;

Description

If EXAplus is connected to the database, this command terminates the current connection. A COMMIT will also be performed if "SET AUTOCOMMIT EXIT" has been set. The command has no effect if there is no connection.

Example(s)

```
DISCONNECT;
```

EXIT and QUIT

Syntax

[EXIT|QUIT][;]

Description

Exit or Quit closes the connection with the database and quits EXAplus (not available for the GUI version). A terminating semicolon is not needed for this.

Example(s)

```
exit;
quit;
```

HOST

Syntax

HOST <command>;

Description

Performs an operating system command on the client host and then returns to EXAplus. Single and double quotes have to be doubled like in SQL strings.



It is not possible to use programs that expect input from the keyboard with the HOST command.

Example(s)

```
host cat test.sql;
```

PAUSE

Syntax

```
PAUSE [<text>];
```

Description

Similar to [PROMPT](#), but waits for the user to press the return key after the text is displayed. Single and double quotes have to be doubled like in SQL strings.

Example(s)

```
prompt 'Please note the following message!';
pause 'Message: &message';
```

PROMPT

Syntax

```
PROMPT [<text>];
```

Description

Prints <text> to the console. Single and double quotes have to be doubled like in SQL strings. If the parameter is not specified, an empty line is output.

Example(s)

```
prompt Ready.;
```

SET AUTOCOMMIT

Syntax

```
SET AUTOCOMMIT ON|OFF|EXIT;
```

Description

Controls whether Exasol should automatically perform COMMIT statements.

- ON After each SQL statement a COMMIT statement is executed automatically. This is preset by default.
- OFF Automatic COMMIT statements are not executed.
- EXIT A COMMIT statement is executed when EXAplus is exited.



The recommended setting for normal applications is "ON". See also [Section 3.1, "Transaction management"](#).

If EXAplus is stopped with CTRL-C, an automatic COMMIT statement is not executed.

Example(s)

```
set autocommit off;
```

SET AUTOCOMPLETION

Syntax

```
SET AUTOCOMPLETION ON|OFF;
```

Description

Switches the auto-completion feature on or off.

Example(s)

```
set autocompletion off;
set autocompletion on;
```

SET COLSEPARATOR

Syntax

```
SET COLSEPARATOR <separator>;
```

Description

Sets the string that separates two columns. Preset to a spacing character.

Example(s)

```
SQL_EXA> set colseparator '|';
SQL_EXA> select * from ty;

      A          |B
-----+-----
      11        |Meier
```

SET DEFINE

Syntax

```
SET DEFINE <C>|ON|OFF;
```

Description

Sets the character with which user variables are initiated. Presetting is the character "&". <C> must relate to one single special character. Via ON and OFF you can activate and deactivate user variables.

Example(s)

```
SQL_EXA> define value=A;
SQL_EXA> prompt &value;
A
SQL_EXA> set define %;
SQL_EXA> prompt %value &value;
A &value
```

SET ENCODING

Syntax

```
SET ENCODING [<encoding>];
```

Description

Sets encoding for different files which EXAplus is reading or writing. Default encoding for reading and writing files is UTF8, but can also be changed via commandline parameter –encoding. For all supported encodings see [Appendix D, Supported Encodings for ETL processes and EXAplus](#). When called without arguments the current settings are listed.

Example(s)

```
set encoding Latin1;
```

SET ESCAPE

Syntax

```
SET ESCAPE <C> | ON | OFF;
```

Description

Specifies the escape character, which makes it possible to input special characters like &. <C> must relate to one single special character. You can activate/deactivate the escape character by using SET ESCAPE ON | OFF. When setting a character, it is activated automatically. In default case, the escape character is \, but it is deactivated.

Example(s)

```
SQL_EXA> define value=a;
SQL_EXA> prompt $&value;
$a
SQL_EXA> set escape $;
SQL_EXA> prompt $&value;
&value
```

SET FEEDBACK

Syntax

```
SET FEEDBACK ON|OFF|<num>;
```

Description

Controls the output of confirmations. This includes the output after statements such as "INSERT INTO" and display of the row number in the output of a table. "OFF" suppresses the output of all confirmations, "ON" switch all confirmations on. If <num> is a numeric value greater than 0, tables with at least <num> rows will have row numbers displayed. The "SET VERBOSE OFF" statement and the command line parameter "-q" sets this setting to "OFF".

Example(s)

```
SQL_EXA> set feedback off;
SQL_EXA> insert into ty values (44,'Schmitt');
SQL_EXA> select * from ty;

A           B
-----
11 Meier
44 Schmitt
```

SET HEADING

Syntax

```
SET HEADING ON|OFF;
```

Description

Switches the output of column headings on or off. Preset to "ON".

Example(s)

```
SQL_EXA> set heading off;
SQL_EXA> select * from ty;
               11 Meier
               44 Schmitt
               45 Huber
               19 Kunze
               87 Mueller
               99 Smith
              125 Dooley
              33 Xiang
              442 Chevaux
```

SET LINESIZE

Syntax

```
SET LINESIZE <n>;
```

Description

Sets width of the output lines to "n" characters. If a table with wider lines is output, EXAplus adds line breaks between two columns.

Example(s)

```
SQL_EXA> set linesize 20;
SQL_EXA> select * from ty;

A
-----
B
-----
               11
Meier
```

SET NULL

Syntax

```
SET NULL <text>;
```

Description

Defines the string for displaying NULL values in tables. Preset to an empty field. This setting is only used for columns that have not had a different representation of NULL values defined with the [COLUMN](#) statement.

Example(s)

```
SQL_EXA> set null EMPTY;
SQL_EXA> select NULL from dual;

NULL
-----
EMPTY
```

SET NUMFORMAT

Syntax

```
SET NUMFORMAT <Formatstring>;
```

Description

Sets the formatting of numeric columns in tables. The format string is the same as that of the [COLUMN](#) statement. There you also can find a closer description of the possible formats. This setting is only used for numeric columns that have not had a different format defined with the [COLUMN](#) statement.

Example(s)

```
SQL_EXA> set numformat 99990.00;
SQL_EXA> select 43 from dual;

43
-----
43.00
```

SET PAGESIZE

Syntax

```
SET PAGESIZE <num> | UNLIMITED;
```

Description

Sets how many text lines there should be before column headings are repeated. Preset to "UNLIMITED", this means that column headings are only displayed before the first line. If the HEADING setting has the value, OFF, no column headings are displayed.

Example(s)

```
SQL_EXA> set pagesize 10;
SQL_EXA> select * from ty;

A          B
-----
11 Meier
44 Schmitt
45 Huber
19 Kunze
87 Mueller
99 Smith
125 Dooley
33 Xiang

A          B
-----
442 Chevaux
```

SET SPOOL ROW SEPARATOR

Syntax

```
SET SPOOL ROW SEPARATOR LF | CR | CRLF | AUTO;
```

Description

Defines the row separator for the **SPOOL** command. The option AUTO (default) uses the common character of the local system.

Example(s)

```
set spool row separator LF;
```

SET TIME

Syntax

```
SET TIME ON|OFF;
```

Description

Switches output of the current time of the client system at the input prompt on or off.

Example(s)

```
SQL_EXA> set time on;
16:28:36 SQL_EXA>
```

SET TIMING

Syntax

```
SET TIMING ON|OFF;
```

Description

Switches display of the time needed for execution of a SQL statement on or off.

Example(s)

```
SQL_EXA> set timing on;
SQL_EXA> create table tx(a DECIMAL);

Timing element: 1
Elapsed: 00:00:00.313
```

SET TRUNCATE HEADING

Syntax

```
SET TRUNCATE HEADING ON|OFF;
```

Description

Defines whether column headings are truncated to the corresponding column data type length. Preset to "ON".

Example(s)

```
SQL_EXA> select * from dual;
D
-
SQL_EXA> set truncate heading off;
SQL_EXA> select * from dual;
DUMMY
-----
```

SET VERBOSE

Syntax

```
SET VERBOSE ON|OFF;
```

Description

Switches additional program information on or off. Preset to ON. The parameter -q is used to set the value to OFF at program startup.

Example(s)

```
set verbose off;
```

SHOW

Syntax

```
SHOW [<var>];
```

Description

Displays the EXAplus settings (see following). If no setting is specified, all are displayed.

Example(s)

```
SQL_EXA> show;
AUTOCOMMIT = "ON"
AUTOCOMPLETION = "ON"
COLSEPARATOR = " "
DEFINE = "&"
ENCODING = "UTF-8"
ESCAPE = "OFF"
FEEDBACK = "ON"
```

```
HEADING = "ON"
LINESIZE = "200"
NULL = "null"
NUMFORMAT = "null"
PAGESIZE = "UNLIMITED"
SPOOL ROW SEPARATOR = "AUTO"
TIME = "OFF"
TIMING = "OFF"
TRUNCATE HEADING = "ON"
VERBOSE = "ON"
```

SPOOL

Syntax

```
SPOOL [<file>|OFF];
```

Description

If a filename is specified as a parameter, the file will be opened and the output of EXAplus saved to this file (the encoding is used which was set via commandline parameter `-encoding` or via the command [SET ENCODING](#)). If the file already exists, it will be overwritten. SPOOL OFF terminates saving and closes the file. By entering SPOOL without parameters, the name of spool file will be displayed.

Example(s)

```
spool log.out;
select * from table1;
spool off;
```

TIMING

Syntax

```
TIMING START|STOP|SHOW;
```

Description

The built-in timer is controlled with this statement.

- | | |
|---------------------|--|
| TIMING START [name] | Starts a new timer with the specified name. If no name is specified, the number of the timer is used as a name. |
| TIMING STOP [name] | Halts the timer with the specified name and displays the measured time. If no name is specified, the timer launched most recently is halted. |
| TIMING SHOW [name] | Displays the measured time from the specified timer without stopping this. If no name is specified, all timers are displayed. |

Example(s)

```
timing start my_timing;
timing show;
timing stop my_timing;
```

UNDEFINE

Syntax

```
UNDEFINE <variable>;
```

Description

Deletes the variable <variable>. If there is no variable with the specified name, an error is reported.

Example(s)

```
undefine message;
```

WHENEVER

Syntax

```
WHENEVER SQLERROR|OSERROR EXIT [<exit_code>] |CONTINUE|ABORT [ROLLBACK|COMMIT];
```

Description

This statement defines the behavior of EXAplus in the event of errors. WHENEVER SQLERROR responds to errors in the execution of SQL statements, WHENEVER OSERROR to operating system errors (file not found, etc.).

The behavior can be specified as follows:

EXIT	Quits EXAplus.
CONTINUE	Continues program operation despite the occurrence of an error.
ABORT	Aborts the execution of statements, but doesn't close EXAplus.



CONTINUE is the default setting for both types of error (and EXIT if EXAplus is started with the parameter -x).

Example(s)

```
whenever sqlerror continue;
whenever sqlerror exit 4;
```

4.2. ODBC driver

This section describes installation, configuration and usage of the **ODBC** driver for Exasol. Further information about the usage of the driver for several products can be found in our Knowledge Center on our homepage wwwexasol.com.

4.2.1. Supported standards

The ODBC driver for Exasol supports the ODBC 3.5 standard (core level).

The following features are not supported:

- "Positioned update/delete" and "batched updates/deletes" (SQLSetPos)
- Asynchronous execution
- Bookmarks

 If the rowcount of a query exceeds $2^{31}-1$, the ODBC driver will return the value 2147483647 for the function SQLRowCount(). The reason for that behavior is the 32-bit limitation for this return value defined by the ODBC standard.

4.2.2. Windows version of the ODBC driver

System requirements

The Exasol ODBC driver is provided for both the 32-bit version and the 64-bit version of Windows. The requirements for installing the ODBC driver are listed below:

- The Windows user who installs the Exasol ODBC and its components must be the computer administrator or a member of the administrator group.
- Microsoft .NET Framework 4.0 Client Profile™ has to be installed on your system.
- All applications and services that integrate the ODBC must be stopped during installation. If "Business Objects XI" is installed on this machine, the "Web Intelligence Report Server" service must be stopped.

The ODBC driver has been successfully tested on the following systems:

- Windows 10 (x86/x64)
- Windows 7, Service Pack 1 (x86/x64)
- Windows Server 2012 R2 (x86/x64)
- Windows Server 2012 (x86/x64)
- Windows Server 2008 R2, Service Pack 1 (x86/x64)
- Windows Server 2008, Service Pack 2 (x86/x64)

Installation

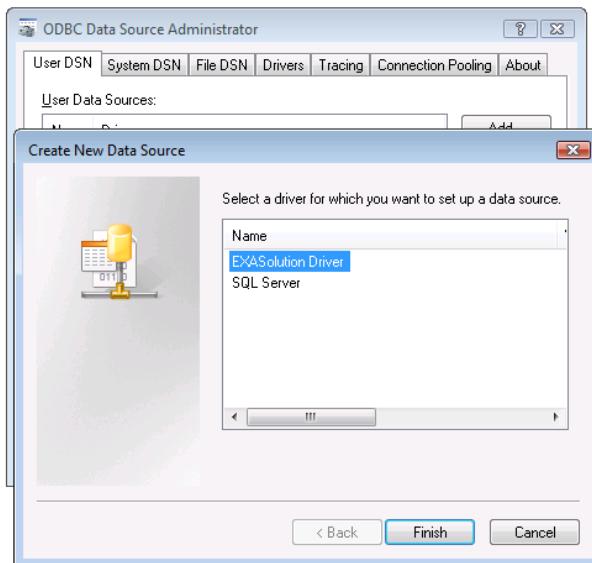
A simple setup wizard leads you through the installation steps after executing the installation file. Please note that separate installation executables are provided for 32bit and 64bit applications.

 Already existing ODBC data sources are automatically configured to the new driver version in case of an update.

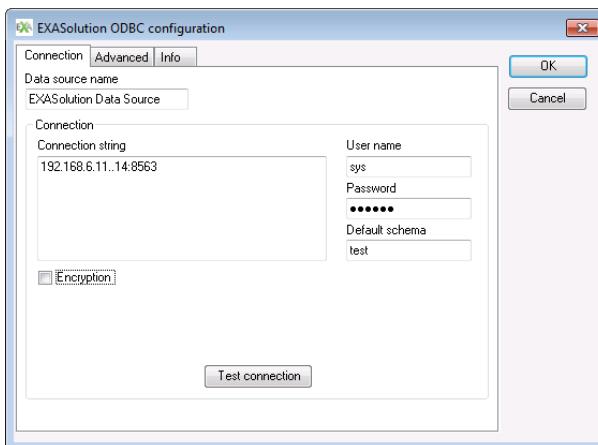
Configuring the driver and the data sources

To configure the ODBC driver you have to use the ODBC configuration tool which is available on Windows systems. A shortcut to that tool can be found in the start menu entry of the Exasol ODBC driver. If both variants are installed (32bit and 64bit), then there exist two shortcuts to the 32bit and 64bit variant of that tool.

Within the tool, please select the "User DSN" or "System DSN" tab. Click on "Add" and select "Exasol Driver" in the window "Create New Data Source".



The Exasol settings are configured in the next step:



Datasource name
Connection string

Name of the new ODBC data source
List of host names or IP addresses and the port of the Exasol cluster (e.g. 192.168.6.11..14:8563).



The computer must be capable of resolving host names. If this is not possible or you are unsure, please contact your network administrator.

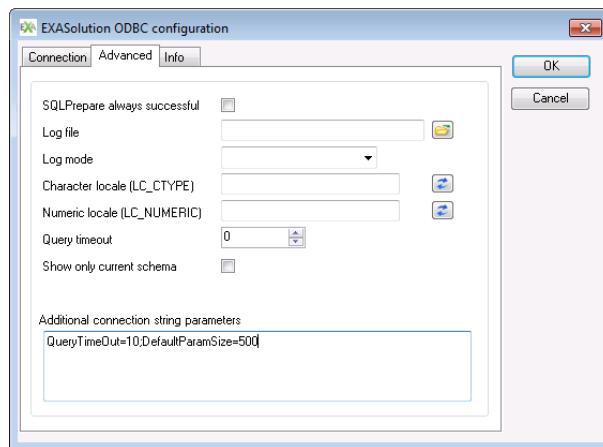
Encryption
User name
Password
Default schema

Defines whether the automatic encryption is switched on or off.
Name of the default user for this connection.
Password of the user.
The name of the schema that should be automatically opened when the connection is established. If this schema is not available during the attempt at establishing the connection, an error message is given and the connection with Exasol is not made.

Test connection

The user can use this to test whether the connection data has been entered correctly.
An attempt to connect with Exasol is made.

In the advanced settings you can define additional options and arbitrary connection string parameters (see also [Section 4.2.5, “Connecting via Connection Strings”](#)):



When you click the "OK" button in the Exasol configuration window, your new connection will appear in the list of Windows data sources.



The Connection Pooling of the driver manager is deactivated by default. You can explicitly activate it in the configuration tool "ODBC Data Source Administrator". But please note that in that case reused connections keep their session settings which were set via SQL commands (see [ALTER SESSION](#)).

Known problems

This chapter covers known problems associated with using the ODBC driver. Generally the causes are easy to correct.

Table 4.3. Known problems associated with using the ODBC driver on Windows

Problem	Solution	Screenshot
System error code: 126, 127, 193 or 14001	Important elements of the ODBC driver are not installed correctly. Quit all applications and services that may be used by the ODBC and reinstall the Exasol ODBC. In case of error 14001, important system libraries needed by the Exasol ODBC are not installed on this machine. Reinstall the Visual C++ Redistributables and afterwards the Exasol ODBC. This error can also be resolved by installing the latest Windows updates.	
Installation error: Incompatible dll versions	A similar message may also occur during the installation. This message occurs if an older version of the Exasol ODBC was already installed and was not overwritten during the installation. Quit all applications and services that may be used by the ODBC and reinstall the Exasol ODBC.	
Error opening file for writing	The installer has detected that it cannot overwrite an Exasol ODBC component. Quit all applications and services that may be used by the ODBC and reinstall the Exasol ODBC.	
Data source name not found and no default driver specified	Please check the Data source name. On 64-bit systems please check whether the created data source is a 64-bit data source, although the application expects a 32-bit data source or vice versa.	

4.2.3. Linux/Unix version of the ODBC driver



For installation and configuration, be sure to read the file `README.txt` which is included in the installation package.



The list of options you can set in the file `odbc.ini` can be found in [Section 4.2.5, “Connecting via Connection Strings”](#).

The Exasol ODBC driver for Linux/Unix has been designed to run on as many distributions as possible. It was successfully tested on the following systems:

- Red Hat / CentOS 7 (x64)

- Red Hat / CentOS 6 (x86/x64)
- Debian 8 (x86/x64)
- Ubuntu 16.04 LTS (x86/64)
- Ubuntu 14.04 LTS (x86/64)
- SUSE Linux Enterprise Server 12 (x64)
- SUSE Linux Enterprise Desktop 12 (x64)
- SUSE Linux Enterprise Server 11 (x86/x64)
- openSUSE Leap 42.2 (x64)

- macOS Sierra (32Bit/64Bit)
- OS X 10.11 (32Bit/64Bit)

- FreeBSD 11.0 (64Bit)
- FreeBSD 10.3 (64Bit)



The ODBC driver was tested on Mac OS X with the driver manager iODBC in two variations (which one is used depends on the application):

- As system library (included in the system)
- As framework in version 3.52.7 (see also www.iodbc.org)

Known problems

Table 4.4. Known problems when using the ODBC driver for Linux/Unix

Description	Solution
Error "Data source name not found, and no default driver specified"	Possibly the unixODBC driver manager uses the wrong <code>odbc.ini</code> . You can set the file which should be used via the environment variable <code>ODBCINI</code> . Moreover a reason could be that the wrong version of unixODBC is used if several versions are installed. You can determine the used version e.g. via <code>isql -version</code> .
Error "Invalid utf-8 character in input"	Set the variable <code>LC_ALL</code> to a locale that is able to display the given characters.
Missing or wrong characters in output of console or your ODBC application	Set the environment variables <code>LC_CTYPE</code> and <code>LC_NUMERIC</code> or the parameters <code>CONNECTIONLCCTYPE</code> and <code>CONNECTIONLCNUMERIC</code> in your data source (<code>odbc.ini</code>) to the locale in which the characters should be displayed. Applications using the DataDirect driver manager normally need a UTF8 locale.

4.2.4. Establishing a connection in an ODBC application

To establish a connection to Exasol via ODBC driver, you can use one of two different ODBC functions:

SQLConnect()

In this method you choose a **DSN** entry and define user and password.

Example call:

```
SQLConnect(connection_handle,
           (SQLCHAR*)"exa_test", SQL_NTS,
```

```
(SQLCHAR*) "sys" , SQL_NTS ,
(SQLCHAR*) "exasol" , SQL_NTS );
```

SQLDriverConnect()

For function `SQLDriverConnect()` there exist two different alternatives. Either you choose a [DSN](#) entry of the file `odbc.ini`, or you choose a certain `DRIVER` from the file `odbcinst.ini`. In both cases a connection string is used to define further options (see next section).

Example call:

```
SQLDriverConnect(
    connection_handle, NULL,
    (SQLCHAR*)"DSN=exa_test;UID=sys;PWD=exasol", SQL_NTS,
    NULL, 0, NULL, SQL_DRIVER_NOPROMPT);

SQLDriverConnect(
    connection_handle, NULL,
    (SQLCHAR*)
    "DRIVER={EXASOL Driver};EXAHOST=192.168.6.11..14:8563;UID=sys;PWD=exasol",
    SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
```

4.2.5. Connecting via Connection Strings

Several key-value pairs are given to a connection function. The driver manager will then load the correct driver and hands over the necessary parameters.

 The data of the connection string have higher priority than the values of the `odbc.ini` file on Linux/Unix systems and the configuration of the data source on Windows systems.

A typical example for a connection string for connecting to Exasol:

```
DSN=exa_test;UID=sys;PWD=exasol;EXASCHEMA=MY_SCHEMA
DRIVER={EXASOL Driver};EXAHOST=192.168.6.11..14:8563;UID=sys;PWD=exasol
```

Currently, the following keys are supported:

EXAHOST Defines the servers and the port of the Exasol cluster (e.g. 192.168.6.11..14:8563).

When opening a connection, the driver will randomly choose an address from the specified address range. If the connection fails the driver will continue to try all other possible addresses.

Instead of a simple IP range you can also define a comma-separated list of host:port pairs.

Examples:

myhost:8563

Single server with name myhost and port 8563.

myhost1,myhost2:8563

Two servers with port 8563.

myhost1..4:8563

Four servers (myhost1, myhost2, myhost3, myhost4) and port 8563.

192.168.6.11..14:8563

Four servers from 192.168.6.11 up to 192.168.6.14 and port 8563 .

	Instead of a concrete list you can also specify a file which contains such a list. (e.g. //c:\\mycluster.txt). The two slashes ("\\") indicate that a filename is specified.
EXAUID or UID	Username for the login. UID is automatically removed from the connection string by some applications.
EXAPWD or PWD	Password of user. PWD is automatically removed from the connection string by some applications.
EXASCHEMA	Schema which is opened directly after the connection.
EXALOGFILE	Log file for the driver.
	 Dependent on the log mode (see below), the log file can grow very quickly. This is not recommended in productive environments.
	 Please mention that the log file can possibly contain sensitive data (SQL commands and data).
LOGMODE	Specifies the mode for the log file. The following values are possible:
DEFAULT	Logs the most important function calls and SQL commands.
VERBOSE	Logs additional data about internal process steps and result data.
ON ERROR ONLY	Only in case of an error the log data is written.
DEBUGCOMM	Extended logs for analyzing the client/server communication (similar to VERBOSE, but without the data and parameter tables).
NONE	No log data is written.
KERBEROSSERVICENAME	Principal name of the Kerberos service. If nothing is specified, the name "exasol" will be used as default.
KERBEROSHOSTNAME	Host name of the Kerberos service. If nothing is specified, the host name of the parameter EXAHOST will be used as default.
ENCRYPTION	Switches on the automatic encryption. Valid values are "Y" and "N" (default is "Y").
AUTOCOMMIT	Autocommit mode of the connection. Valid values are "Y" and "N" (default is "Y").
QUERYTIMEOUT	Defines the query timeout in seconds for a connection. If set to 0, the query timeout is deactivated. Default is 0.
CONNECTTIMEOUT	Maximal time in milliseconds the driver will wait to establish a TPC connection to a server. This timeout is interesting to limit the overall login time especially in case of a large cluster with several reserve nodes. Default: 2000
SUPERCONNECTION	Enables the user to execute queries even if the limit for active sessions (executing a query) has been reached. Valid values are "Y" and "N" (default is "N").



This parameter can be set only by user SYS.



SUPERCONNECTION should only be used only in case of significant performance problems where it is not possible anymore to log in and execute queries within a reasonable time. By that parameter you can e.g. analyze the system and kill certain processes which cause the problem.

PREPAREAS	Prepare always successful: SQLPrepare and SQLPrepareW always return SQL_SUCCESS. Valid values: "Y" or "N" (default "N").
MAXPARAMSIZE	Maximal size for VARCHAR parameters in prepared statements, even if the determined data type was bigger. Use value 0 for deactivating. Default is 0.
DEFAULTPARAMSIZE	Default size for VARCHAR parameters in prepared statements whose type cannot be determined at prepare time. Default is 2000.
COGNOSUPPORT	If you want to use the Exasol ODBC driver in combination with Cognos, we recommend to set on this option on "Y". Default is "N".
CONNECTIONLCCTYPE	Sets LC_CTYPE to the given value during the connection. Example values for Windows: "deu", "eng". Example values for Linux/Unix: "de_DE", "en_US". On Linux/Unix, you can also set an encoding, e.g. "en_US.UTF-8".
CONNECTIONLCNUMERIC	Sets LC_NUMERIC to the given value during the connection. Example values for Windows: "deu", "eng". Example values for Linux/Unix: "de_DE", "en_US".
SHOWONLYCURRENTSCHEMA	Defines whether the ODBC driver considers all schemas or just the current schema for metadata like the list of columns or tables. Valid values are "Y" and "N" (default is "N").
STRINGSNOTNULL	Defines whether the ODBC driver returns empty strings for NULL strings when reading table data (please notice that the database internally doesn't distinguish between NULL and empty strings and returns in both cases NULL values to the driver). Valid values are "Y" and "N" (default is "N").
INTTYPESINRESULTSFPOSSIBLE	If you switch on this option, then DECIMAL types without scale will be returned as SQL_INTEGER (9 digits) or SQL_BIGINT (18 digits) instead of SQL_DECIMAL. Valid values are "Y" and "N" (default is "N").

4.2.6. Support for character sets

Internally, the ODBC driver uses the unicode character set UTF8. Other character set input data is converted to UTF8 and transferred to Exasol. Data from Exasol is converted by the ODBC driver into the character set which is specified on the client site.

On Windows servers the data which is sent to the ODBC driver has to be in an encoding which is installed locally. The encoding can be specified via the language settings of the connection (see above).

On Linux/Unix servers, the encoding can be set via environment variables. To set e.g. the German language and the encoding UTF8 in the console, the command `export LC_CTYPE=de_DE.utf8` can be used. For graphical applications, a wrapper script is recommended.

It is important that the used encoding has to be installed on your system. You can identify the installed encodings via the command "locale -a".

4.2.7. Best Practice for developers

Reading big data volumes

Never use `SQLGetData`, but `SQLBindCol` and `SQLFetch`.

To achieve the best performance you should try to fetch about 50-100 **MB** of data by choosing the number of rows per `SQLFetch`.

Inserting data into the database

Instead of using single insert statements like "INSERT INTO t VALUES 1, 2, ..." you should use the more efficient interface of prepared statements and their parameters. Prepared statements achieve optimal performance when using parameter sets between 50 and 100 **MB**.

Moreover you should insert the data by using the native data types. E.g. for number 1234 you should use `SQL_C_SLONG` (Integer) instead of `SQL_CHAR` ("1234").

Autocommit mode

Please note that you should deactivate the autocommit mode on Windows systems only via the method `SQLSetConnectAttr()` and not in the data source settings. Otherwise, the windows driver manager doesn't notice this change, assumes that autocommit is switched on and doesn't pass `SQLEndTran()` calls to the database. This behavior could lead to problems.

Problems using the localizaton attributes

If the settings `CONNECTIONLCCTYPE` or `CONNECTIONLCNUMERIC` do not work, the connect function will return `SQL_SUCCESS_WITH_INFO` and a corresponding text is added in [DiagRec](#).

4.3. JDBC driver

Exasol provides a [JDBC](#) driver for connecting third party applications to Exasol.

4.3.1. Supported standards

The JDBC driver, which is made available by Exasol for Exasol, supports the JDBC 3.0 Core [API](#). This standard includes:

- Access to the driver via the DriverManager-API and configuration of the driver using the DriverPropertyInfo interface
- Execution of SQL statements directly, as a prepared statement, and in batch mode
- Support of more than one open ResultSet
- Support of the following metadata APIs: DatabaseMetaData and ResultSetMetaData

The following features are not supported:

- Savepoints
- User-defined data types and the types Blob, Clob, Array and Ref.
- Stored procedures
- Read-only connections
- The API ParameterMetaData



If the rowcount of a query exceeds $2^{31}-1$, the JDBC driver will return the value 2147483647. The reason for that behavior is the 32-bit limitation for this return value defined by the JDBC standard.



Detailed information about the supported interfaces are provided in the *API Reference* which you can find either in the start menu (Windows) or in the folder `html` of the install directory (Linux/Unix).

4.3.2. System requirements

The JDBC driver requires a Java environment ([JRE](#)) from Sun Java 1.5 onwards. Proper functioning of the driver cannot be guaranteed in combination with other JREs. The driver itself can be used on any platform. That is, it can be used on any platform for which a suitable [JRE](#) is available.

The driver has been successfully tested on the following systems:

- Windows 10 (x86/x64)
- Windows 7, Service Pack 1 (x86/x64)
- Windows Server 2012 R2 (x86/x64)
- Windows Server 2012 (x86/x64)
- Windows Server 2008 R2, Service Pack 1 (x86/x64)
- Windows Server 2008, Service Pack 2 (x86/x64)
- Red Hat / CentOS 7, OpenJDK JVM 1.8.0 (x64)
- Red Hat / CentOS 6, OpenJDK JVM 1.8.0 (x86/x64)
- Debian 8, OpenJDK JVM 1.7.0 (x86/x64)
- Ubuntu 16.04 LTS, OpenJDK JVM 1.8.0 (x86/64)
- Ubuntu 14.04 LTS, OpenJDK JVM 1.7.0 (x86/64)
- SUSE Linux Enterprise Server 12, IBM's JVM 1.7.0 (x64)
- SUSE Linux Enterprise Desktop 12, OpenJDK JVM 1.7.0 (x64)
- SUSE Linux Enterprise Server 11 Service Pack 3, IBM's JVM 1.7.0 (x86/x64)
- openSUSE Leap 42.2, OpenJDK JVM 1.8.0 (x64)

- macOS Sierra, JVM 1.8.0 (64Bit)
- OS X 10.11, JVM 1.8.0 (64Bit)
- FreeBSD 11.0, OpenJDK 1.8.0 (64Bit)
- FreeBSD 10.3, OpenJDK 1.8.0 (64Bit)

For Windows systems an automatic installation wizard exists. In this case, Microsoft .NET Framework 4.0 Client Profile™ has to be installed on your system.

4.3.3. Using the driver

Installing JDBC directly

On Windows systems, the driver is installed as a .jar archive in the installation directory. On Linux/Unix systems the driver is included in the delivered .tgz file. Depending on the application this archive must be added to the search path for the Java classes (CLASSPATH).

All classes of the JDBC driver belong to the Java "com.exasol.jdbc" package. The main class of the driver is

```
com.exasol.jdbc.EXADriver
```

Integrating JDBC by using Maven

The JDBC driver is also available in the Exasol Maven repository (<https://mavenexasol.com>). Please add the following repository and dependency to the build configuration of your project (e.g. pom.xml for Maven):

```
<repositories>
  <repository>
    <id>maven.exasol.com</id>
    <url>https://mavenexasol.com/artifactory/exasol-releases</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>com.exasol</groupId>
    <artifactId>exasol-jdbc</artifactId>
    <version>6.0.0</version>
  </dependency>
</dependencies>
```

URL of Exasol

The URL for the Exasol used by the driver has the following form:

```
jdbc:exa:<host>:<port>[ ;<prop_1>=<value_1>]...[ ;<prop_n>=<value_n> ]
```

Meaning of the individual elements of the URL:

<i>jdbc:exa</i>	This prefix is necessary for the driver manager.
<i><host>:<port></i>	Defines the servers and the port of the Exasol cluster (e.g. 192.168.6.11..14:8563).

When opening a connection, the driver will randomly choose an address from the specified address range. If the connection fails the driver will continue to try all other possible addresses.

Instead of a simple IP range you can also define a comma-separated list.

Examples:

myhost:8563	Single server with name myhost and port 8563.
myhost1,myhost2:8563	Two servers with port 8563.
myhost1..4:8563	Four servers (myhost1, myhost2, myhost3, myhost4) and port 8563.
192.168.6.11..14:8563	Four servers from 192.168.6.11 up to 192.168.6.14 and port 8563 .

<prop_i=value_i>

Instead of a concrete list you can also specify a file which contains such a list. (e.g. //c:\\mycluster.txt). The two slashes ("\\") indicate that a filename is specified. An optional list of properties separated by a ";" follows the port, the values of which should be set when logging-in. These properties correspond with the supported Driver Properties and are described in the following section. It is important to note that the values of properties within the URL can only consist of alphanumeric characters.

Example from a Java program.

```

import java.sql.*;
import com.exasol.jdbc.*;

public class jdbcsample
{
    public static void main(String[] args)
    {
        try {
            Class.forName("com.exasol.jdbc.EXADriver");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        Connection con=null;
        Statement stmt=null;
        try {
            con = DriverManager.getConnection(
                "jdbc:exa:192.168.6.11..14:8563;schema=SYS",
                "sys",
                "exasol"
            );
            stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM CAT");
            System.out.println("Schema SYS contains:");
            while(rs.next())
            {
                String str1 = rs.getString("TABLE_NAME");
                String str2 = rs.getString("TABLE_TYPE");
                System.out.println(str1 + ", " + str2);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {stmt.close();} catch (Exception e) {e.printStackTrace();}
            try {con.close();} catch (Exception e) {e.printStackTrace();}
        }
    }
}

```

```
        }  
    }  
}
```

The above code sample builds a JDBC connection to Exasol, which runs on servers 192.168.6.11 up to 192.168.6.14 on port 8563. User `sys` with the password `exasol` is logged-in and the schema `sys` is opened. After that all tables in this schema is shown:

```
Schema SYS contains:  
EXA_SQL_KEYWORDS, TABLE  
DUAL, TABLE  
EXA_PARAMETERS, TABLE  
...
```

Supported DriverProperties

The following properties can be transferred to the JDBC driver via the URL:

Table 4.5. Supported DriverProperties of the JDBC driver

Property	Values	Description
schema	String	Name of the schema that should be opened after login. If the schema cannot be opened, the login fails with a java.sql.SQLException. Default: no schema to open
autocommit	0=off, 1=on	Switches autocommit on or off. Default: 1
encryption	0=off, 1=on	Switches automatic encryption on or off. Default: 1
kerberoservice-name	String	Principal name of the Kerberos service. If nothing is specified, the name "exasol" will be used as default.
kerberoshostname	String	Host name of the Kerberos service. If nothing is specified, the host name of the connections string will be used as default.
fetchsize	numeric, >0	<p>Amount of data in kB which should be obtained by Exasol during a fetch. The JVM can run out of memory if the value is too high. Default: 2000</p> <p> The same can be achieved by using the function <code>setFetchSize()</code>.</p>
debug	0=off, 1=on	<p>Switches on the driver's log function. The driver then writes for each established connection a log file named <code>jdbc_timestamp.log</code>. These files contain information on the called driver methods and the progress of the JDBC connection and can assist the Exasol Support in the diagnosis of problems.</p> <p> Due to performance reasons the logging should not be used in a productive system.</p> <p>Default: 0</p>
logdir	String	<p>Defines the directory where the JDBC debug log files shall be written to (in debug mode). Example: <code>jdbc:exa:192.168.6.11..14:8563;debug=1;log-dir=/tmp/my_folder/;schema=sys</code></p> <p> Default is the application's current directory, which is not always transparent. That is why you should always set the log directory in debug mode.</p>
clientname	String	Tells the server what the application is called. Default: "Generic JDBC client"
clientversion	String	Tells the server the version of the application. Default: empty ("")
logintimeout	numeric, >=0	Maximal time in seconds the driver will wait for the database in case of a connect or disconnect request. Default is 0 (unlimited)
connecttimeout	numeric, >=0	Maximal time in milliseconds the driver will wait to establish a TPC connection to a server. This timeout is interesting to limit the overall login time especially in case of a large cluster with several reserve nodes. Default: 2000
querytimeout	numeric, >=0	Defines how many seconds a statement may run before it is automatically aborted. Default is 0 (unlimited)

Property	Values	Description
superconnection	0=off, 1=on	<p>Enables the user to execute queries even if the limit for active sessions (executing a query) has been reached.</p> <p> This parameter can be set only by user SYS.</p> <p> superconnection should only be used only in case of significant performance problems where it is not possible anymore to log in and execute queries within a reasonable time. By that parameter you can e.g. analyze the system and kill certain processes which cause the problem.</p> <p>Default: 0</p>
slave	0=off, 1=on	<p>So-called sub-connections for parallel read and insert have this flag switched on. Details and examples can be found in our Solution Center: https://wwwexasolcom/support/browse/SOL-546</p> <p>Default: 0</p>
slavetoken	numeric, >=0	<p>Is necessary to establish parallel sub-connections. Default: 0</p>

4.3.4. Best Practice for developers

Memory of **JVM**

When operating on big tables, you can get problems if the specified memory for the **JVM** is not big enough.

Reading big data volumes

Via the parameter "fetchsize", you can determine the data volume which should be fetched from the database per communication round. If this value is too low, the data transfer can take too long. If this value is too high, the **JVM** can run out of memory. We recommend a fetch size of 1000-2000.

Inserting data into the database

Instead of using single insert statements like "INSERT INTO t VALUES 1, 2, ..." you should use the more efficient interface of prepared statements and their parameters. Prepared statements achieve optimal performance when using parameter sets between 500 **KB** and 20 **MB**. Moreover you should insert the data by using the native data types.

Unused resources

Unused resources should be freed immediately, e.g. Prepared Statements via "close()".

Connection servers

Don't specify a needless wide IP address range. Since those addresses are randomly tried until a successful connection, the "connect" could take much time.

4.4. ADO.NET Data Provider

A ADO.NET Data Provider is provided for connecting .NET applications to Exasol.

4.4.1. Supported standards, system requirements and installation

The Exasol data provider supports the ADO.NET 2.0 standard and has been successfully tested on the following systems:

- Windows 10 (x86/x64)
- Windows 7, Service Pack 1 (x86/x64)
- Windows Server 2012 R2 (x86/x64)
- Windows Server 2012 (x86/x64)
- Windows Server 2008 R2, Service Pack 1 (x86/x64)
- Windows Server 2008, Service Pack 2 (x86/x64)



Microsoft .NET Framework 4.0 Client Profile™ has to be installed on your system.

The Data Provider is available for download in the form of an executable installer file. The installation requires administrator rights and is started by double-clicking on the downloaded file. The Data Provider is installed in the global assembly cache and registered in the global configuration file of the .NET framework.



Besides the ADO.NET driver, the additional tool Data Provider Metadata View ™(DPMV) is installed with which you can easily check the connectivity and run simple metadata queries.

Necessary extensions are automatically installed for the following software:

SQL Server Integration Services	Installation of the Data Destination as a data flow destination for Integration Services projects for: <ul style="list-style-type: none">• Visual Studio 2005 with SQL Server 2005 (v 9.0)• Visual Studio 2008 with SQL Server 2008 (v 10.0)• Newer versions work with generic ADO.NET and ODBC and don't need a Data Destination
SQL Server Reporting Services	Installation of the Data Processing Extension and adjustment of the configuration of the report server for: <ul style="list-style-type: none">• Visual Studio 2005 with SQL Server 2005 (v 9.0)• Visual Studio 2008 with SQL Server 2008 (v 10.0)• Visual Studio 2010 with SQL Server 2012 (v 11.0)• Visual Studio 2012 with SQL Server 2012 (v 11.0)• Visual Studio 2013 with SQL Server 2014 (v 12.0)
SQL Server Analysis Services	Installation of the DDEX provider and pluggable SQL cartridge for: <ul style="list-style-type: none">• Visual Studio 2008 with SQL Server 2008 (v 10.0)• Visual Studio 2010 with SQL Server 2012 (v 11.0)• Visual Studio 2012 with SQL Server 2012 (v 11.0)• Visual Studio 2013 with SQL Server 2014 (v 12.0)• Visual Studio 2015 with SQL Server 2016 (v 13.0)• Visual Studio 2013 with SQL Server 12.0

4.4.2. Using the Data Provider

In order to use the Data Provider with a client application, the Data Provider must first be selected in the application. This is usually done by selecting the entry "Exasol Data Provider" from the list of installed Data Providers, or by

entering an *invariant* identifier, with which the Data Provider is selected. The identifier of the Exasol Data Provider is "Exasol.EXADataProvider".

In order to connect with Exasol, the Data Provider must be given a connection string containing any information necessary for establishing a connection from the client application. The connection string is a sequence of keyword/value pairs separated by semicolons. A typical example of a connection string for the Exasol Data Provider is:

```
host=192.168.6.11..14:8563;UID=sys;PWD=exasol;Schema=test
```

The Exasol Data Provider supports the following keywords:

Table 4.6. Keywords in the ADO.NET Data Provider connection string

Keyword	Meaning
server or host	Defines the servers and the port of the Exasol cluster (e.g. 192.168.6.11..14:8563).
port	Port of Exasol. This port is used if you did not specify any port within the parameter host.
user id, username or uid	Username
password or pwd	Password
schema	Schema to be opened on login.
autocommit	Settings for autocommit: ON or OFF. ON by default.
encryption	Settings for the automatic encryption: ON or OFF. ON by default.
logfile	Log file for the driver (e.g. <code>logfile='C:\tmp\ado.log'</code>). Since this log file can grow very fast, we recommend to switch on this option only to analyze problems.
onconnect	SQL string which is executed directly after the connection has been established. If an error occurs, the connection will be aborted.
connecttimeout	Maximal time in milliseconds the driver will wait to establish a TPC connection to a server. This timeout is interesting to limit the overall login time especially in case of a large cluster with several reserve nodes. Default: 2000
querytimeout	Defines how many seconds a statement may run before it is automatically aborted. Default is 0 (unlimited)
superconnection	Enables the user to execute queries even if the limit for active sessions (executing a query) has been reached. Value: ON or OFF. OFF by default.
	 This parameter can be set only by user SYS.  superconnection should only be used only in case of significant performance problems where it is not possible anymore to log in and execute queries within a reasonable time. By that parameter you can e.g. analyze the system and kill certain processes which cause the problem.

A Code example for Exasol Data Provider (written in C#):

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.Common;

namespace ConnectionTest
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            DbProviderFactory factory=null;  
            try  
            {  
                factory = DbProviderFactories.GetFactory("Exasol.EXADataProvider");  
                Console.WriteLine("Found Exasol driver");  
  
                DbConnection connection = factory.CreateConnection();  
                connection.ConnectionString =  
                    "Server=192.168.6.11..14;Port=8563;UID=sys;PWD=exasol;Schema=sys";  
  
                connection.Open();  
                Console.WriteLine("Connected to server");  
                DbCommand cmd = connection.CreateCommand();  
                cmd.Connection = connection;  
                cmd.CommandText = "SELECT * FROM CAT";  
  
                DbDataReader reader = cmd.ExecuteReader();  
  
                Console.WriteLine("Schema SYS contains:");  
                while (reader.Read())  
                {  
                    Console.WriteLine("{0}, {1}",  
                        reader[ "TABLE_NAME" ],  
                        reader[ "TABLE_TYPE" ]);  
                }  
  
                reader.Close();  
                connection.Close();  
            }  
            catch (Exception ex)  
            {  
                Console.WriteLine(ex.ToString());  
            }  
        }  
    }  
}
```

If this example is executed successfully, the following output will be produced:

```
Found Exasol driver  
Connected to server  
Schema SYS contains:  
EXA_SQL_KEYWORDS, TABLE  
DUAL, TABLE  
EXA_PARAMETERS, TABLE  
...
```

Schema Collections

The Exasol ADO.NET driver supports the following Schema Collections:

- MetaDataCollections

- DataSourceInformation
- DataTypes
- Restrictions
- ReservedWords
- Tables
- Columns
- Procedures
- ProcedureColumns
- Views
- Schemas

Best Practice for developers

• Inserting data into the database

Instead of using single insert statements like "INSERT INTO t VALUES 1, 2, ..." you should use the more efficient interface of prepared statements and their parameters. Prepared statements achieve optimal performance when using parameter sets between 500 kB and 20 MB. Moreover you should insert the data by using the native data types. For execution of prepared statements we recommend to use the interface "IParameterTable". An instance of "ParameterTable" can be created with a "Command" via the method "CreateParameterTable()".

• Block-wise Update by using Prepared Statements

Exasol ADO.NET provides a special interface to execute prepared statements (class ParameterTable) since it is not possible to update rows in Exasol via the ADO.NET specific interfaces DataSet and DataAdapter.

Executing parametrized queries is done by using SQL statements including question marks at the position of parameters. The values of the ParameterTable are used to replace the parameters when executing the statement. Example for a parametrized query:

```
using (EXACommand command = (EXACommand)connection.CreateCommand())
{
    command.CommandText = "CREATE OR REPLACE TABLE TEST (I INT, TS TIMESTAMP)";
    command.ExecuteNonQuery();
    command.CommandText = "INSERT INTO TEST VALUES (?, ?)";
    command.Prepare();
    IParameterTable tab = command.CreateParameterTable();
    tab.AddColumn(DbType.Int32);
    tab.AddColumn(DbType.String);
    for (int i = 0; i < 10; i++)
    {
        int currentRow = tab.AppendRow();
        tab[0].setInt32(currentRow, i);
        tab[1].setString(currentRow, "2017-06-30 11:21:13." + i);
    }
    command.Execute(tab);
}
```



For specifying decimal types, two possible options exist:

- **AddColumn(DbType.Decimal):** When using this general method, the internal type DECIMAL(36,8) is used.
- **AddDecimalColumn(p,s):** When using this specific method, the internal type DECIMAL(p,s) is used (36 digits as maximum).

• Decimal types

The ADO.NET DECIMAL data type can only store values with a precision of a maximum of 28 digits. Decimal columns in Exasol with greater precision do not fit in these data types and have to be provided as strings.

- **Column names**

Please consider the case when using identifiers.

- **Batch execution**

If you want to execute several statements in a batch, you can use the functions `AddBatchCommandText()` and `ExecuteBatch()` of the class `EXACommand`. Example for a batch execution:

```
using (EXACommand cmd = (EXACommand)connection.CreateCommand())
{
    cmd.AddBatchCommandText("CREATE OR REPLACE TABLE TEST (I INT)");
    cmd.AddBatchCommandText("INSERT INTO TEST VALUES (1)");
    cmd.AddBatchCommandText("SELECT * FROM TEST");
    DbDataReader reader = cmd.ExecuteBatch();
    do
    {
        if (reader.RecordsAffected>0)
            Console.Out.WriteLine("Records affected: " + reader.RecordsAffected);
    } while (reader.NextResult());
}
```

4.4.3. Exasol Data Destination

Exasol Data Destination implements the interface "PipelineComponent" for Microsoft SQL Server Integration Services. Data Destination is used to insert data into Exasol from any data flow source.

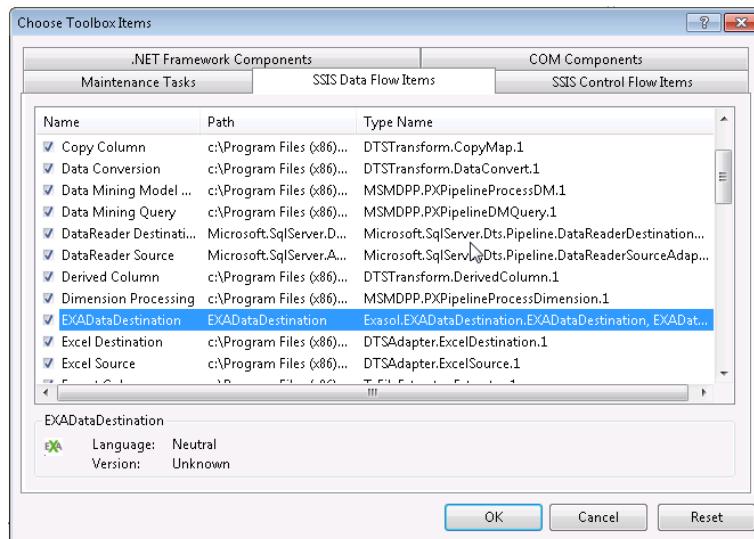
Creating a DataReaderSource

- Create a new "Integration Services Project" in Visual Studio.
- Create a new data flow and insert a "Data Reader Source" from the toolbox section "data flow sources" in the data flow.
- Create a new "ADO.NET Connection" in the "Connection Manager" and choose the Exasol Data Provider from the list of .NET providers.
- Configure the Exasol Data Provider and then test the connection.
- Open the new "Data Reader Source" from the data flow via double-clicking.

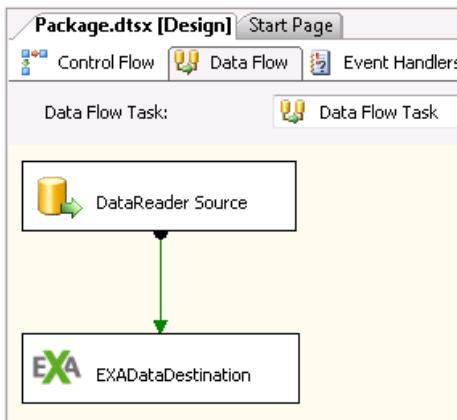
Connection Manager	Select the created connection
Component Properties	In section "User-defined Properties", enter the SQL that will read the data from Exasol (e.g. "select * from my_schema.my_table").
Column Assignments	View and configure the column assignments.
I/O Attributes	Adapt column names and data types.

Adding and using a EXADestination

- In the toolbox, open the context menu of the element "Data Flow Destinations" via a right mouse-click and click "Select Elements ...".
- Select EXADestination in order to make the Exasol Data Destination available in the toolbox.



- Create an instance of the EXADestination by using the mouse to drag the object from the toolbox to the workspace.
- Connect the output of the data source (green) with the EXADestination instance.



- The EXADestination configuration dialog can be opened by a double-click.

Connection Manager

Select a valid Exasol ADO.NET connection.

Component Properties

Exasol specific properties of the data flow destination:

- **AppendTable:** Option to be able to insert data into existing tables
- **Create or replace table:** Option to replace a table if it already exists
- **Insert columns by name:** Use this option if the order of the columns differ between source and destination
- **Log file name:** Name of the log file for debugging purposes
- **TableName:** Name of the target table within Exasol (can also be schema-qualified, e.g. my_schema.my_table).

Command Columns

Defines the columns that will actually be transferred.

I/O Properties

Here you can define further options per column, e.g. the names and data types.



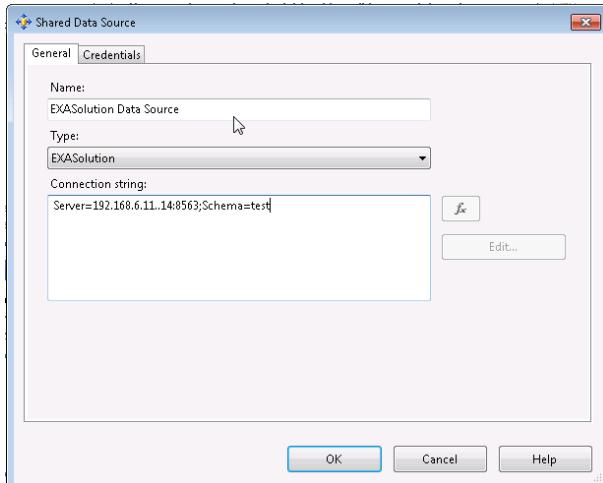
Changing the data type can be very useful if the original data type is decimal with a precision greater than 28. Such data can be transferred to Exasol by altering the data type to e.g. "varchar (40)".

- Once the data flow destination is configured, the data flow can be tested or started. If the sequence is successful, the data flow elements turn green.

4.4.4. Exasol Data Processing Extension

The EXADataProcessingExtension implements interfaces for Microsoft SQL Server Reporting Services in order to create reports with Exasol. In the following example Visual Studio 2005 is used to create a report for the Reporting Services:

- Start Visual Studio 2005 create a new report server project and Open the Project sheet explorer.
- Create a new data source (EXADatasource) and insert the connection string, user and password (see tab "Credentials")



- In the Project sheet explorer, add a new report using the wizard. When selecting the data source, choose the ExaDataSource that was previously configured.
- As soon as the report is ready, it must be created and published. Both functions are contained in the "Create" menu.
- The report can now be run in your browser. To do this, open the browser (e.g. Internet Explorer) and enter the address of the report server (e.g. "http://localhost/Reports"). Check with your system administrator that you have the necessary permissions for running reports in the report server.
- After being run, the output in the browser could look like this:

TABLE TYPE	TABLE NAME
TABLE	BO_CURR_EX
TABLE	CHARTS
TABLE	PROD

4.5. WebSockets

The **JSON** over WebSockets client-server protocol allows customers to implement their own drivers for all kinds of platforms using a connection-based web protocol. The main advantages are flexibility regarding the programming languages you want to integrate Exasol into, and a more native access compared to the standardized ways of communicating with a database, such as **JDBC**, **ODBC** or **ADO.NET**, which are mostly old and static standards and create additional complexity due to the necessary driver managers.

If you are interested to learn more about our WebSockets API, please have a look at our open source GitHub repository (<https://www.github.com/exasol>) where you'll find a lot of details about the API specification and example implementations such as a native Python driver based on the protocol. And we would be very happy if you would contribute to our open source community by using, extending and adding to our open-sourced tools.

4.6. SDK

Exasol provides the SDK (Software Development Kit) to connect client applications with Exasol. You'll find the package in the download area on our website which includes the following subfolders:

R and Python	R- and Python packages for easily executing Python and R code on the Exasol database within your usual programming environment. More details can be found in the corresponding readme files.
CLI	Call level interface for developing C++ applications. More details can be found in the following section.

4.6.1. The call level interface (CLI)

The native C++ interface

The native interface (Exasol **CLI**) implements the majority of the ODBC variation of the Call Level Interfaces Standard. The interface functions are very similar to the ODBC functions; in most cases, the only distinguishing factor is the prefix "EXA".

Notes on the Exasol CLI interface:

- Exasol CLI is faster than ODBC and JDBC
- Special Exasol functionality is supported (e.g. batch execution, parallel reading, parallel insertion)
- The Exasol CLI Library is available for Windows and Linux
- Every ODBC programmer will become familiar with the Exasol CLI Interface very quickly

CLI for Windows

Delivery scope and system requirements

In the CLI subfolder, the following is contained:

- Exasol CLI Libraries: EXACLI.lib and all dynamic libraries (*.dll) in the directory "lib32" or "lib64", compiled for 32 and 64-bit Windows.
- The header files "exaCInterface.h" and "exaDefs.h". These contain declarations and constants of the Exasol CLI Interface.
- A fully functional sample program, which illustrates integration of the CLI interface. The sample is supplied as source code (.cpp). Additionally, the package contains a suitable project file for Visual Studio 10 (.vcproj).

The CLI can be used on 32-bit and 64-bit Windows versions and has been successfully tested on the following systems:

- Windows 10 (x86/x64)
- Windows 7, Service Pack 1 (x86/x64)
- Windows Server 2012 R2 (x86/x64)
- Windows Server 2012 (x86/x64)
- Windows Server 2008 R2, Service Pack 1 (x86/x64)
- Windows Server 2008, Service Pack 2 (x86/x64)



Microsoft .NET Framework 4.0 Client Profile™ has to be installed on your system.

Translating and running the Windows sample program

You need Visual C++, Visual Studio 10 or newer for this.

Open the project file (.vcproj) from the CLI in the directory "examples\sqlExec\" with Visual Studio. The sample sources (.cpp) are already integrated in the project. The Lib and include paths point to the corresponding directories, which were installed with the CLI. If files from the CLI package have been moved or changed, the project must be adapted accordingly.

The sample program can be compiled for 32 and 64-bit as a debug and release version. The generated .exe file is a standard Windows console application. The source code of the sample program contains comments, which describe the parts of the application.

A call to the sample program, "sqlExec.exe", could be something like this:

```
echo select * from exa_syscat | sqlExec.exe -execDirect -u sys -P exasol  
-c 192.168.6.11..14:8563
```

This submits the SQL string "select * from exa_syscat" via standard-in to the sample program and via the CLI function, "EXAExecDirect()", it is executed in Exasol.

CLI for Linux/Unix

Delivery scope



For configuration, be sure to read the file README.txt which is included in the installation package.

The CLI includes:

- Exasol CLI Libraries: dynamic libraries (`libexacli-u02212.so` and `libexacli-u02214.so`), following the unixODBC versions 2.2.12 and 2.2.14 and compiled with gcc 4.1.0.
- The header files "exaCInterface.h" and "exaDefs.h". These contain the declarations and constants of the Exasol CLI Interface.
- A fully functional sample program, which illustrates integration of the CLI interface. The sample is supplied as source code (.cpp) with an appropriate Makefile.

System requirements

The Exasol CLI has been successfully tested on the following systems:

- Red Hat / CentOS 7 (x64)
- Red Hat / CentOS 6 (x86/x64)
- Debian 8 (x86/x64)

- Ubuntu 16.04 LTS (x86/64)
- Ubuntu 14.04 LTS (x86/64)
- SUSE Linux Enterprise Server 12 (x64)
- SUSE Linux Enterprise Desktop 12 (x64)
- SUSE Linux Enterprise Server 11 (x86/x64)
- openSUSE Leap 42.2 (x64)

- FreeBSD 11.0 (64Bit)
- FreeBSD 10.3 (64Bit)

Compiling and running the Linux sample program

You can compile the example program using the Makefile in the subfolder `examples`. GCC 4.1.0 or higher is recommended for that.

A call to the sample program, "exaexec", could be something like this:

```
echo "select * from exa_syscat" | ./exaexec -execDirect -u sys -P exasol
                                         -c 192.168.6.11..14:8563
-----
RESULTCOLS=3
ROWS=83
PARAMCOUNT=0
```

This example executes the given sql statement (`select * from exa_syscat`) and returns the number of columns and rows of the result.

Example

The following example illustrates how to send SQL queries to the database. The return values are not checked to simplify the code.

```
// Like in the ODBC interface, handles are created before the connection
// is established (for environment and connection)
SQLHENV henv;
SQLHDBC hdbc;
EXAAllocHandle(SQL_HANDLE_ENV, NULL, &henv);
EXAAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

// Connect to a cluster with 3 nodes
EXAServerConnect(hdbc, "192.168.6.11..14:8563", SQL_NTS, NULL,
                  0, "sys", SQL_NTS, "exasol", SQL_NTS);

// Create a SQL statement handle
SQLHSTMT hstmt;
EXAAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

// Create schema "TEST"
EXAExecDirect(hstmt, (SQLCHAR*)"CREATE SCHEMA test", SQL_NTS);

// Free the resources and close the connection
EXAFreeHandle(SQL_HANDLE_STMT, hstmt);
EXADisconnect(hdbc);
EXAFreeHandle(SQL_HANDLE_DBC, hdbc);
EXAFreeHandle(SQL_HANDLE_ENV, henv);
```

Best Practice for developers

Reading big data volumes

Never use `SQLGetData`, but `SQLBindCol` and `SQLFetch`.

To achieve the best performance you should try to fetch about 50-100 **MB** of data by choosing the number of rows per `SQLFetch`.

Inserting data into the database

Instead of using single insert statements like "INSERT INTO t VALUES 1, 2, ..." you should use the more efficient interface of prepared statements and their parameters. Prepared statements achieve optimal performance when using parameter sets between 50 and 100 **MB**.

Moreover you should insert the data by using the native data types. E.g. for number 1234 you should use `SQL_C_SLONG` (Integer) instead of `SQL_C_CHAR ("1234")`.

Appendix A. System tables

A.1. General information

Exasol disposes of numerous system tables, which describe the metadata of the database and the current status of the system. These system tables are located in the "SYS" schema but are automatically integrated into the current namespace. This means that if an object with the same name does not exist in the current schema, they can be queried without stating the schema name, "SYS". Otherwise, the system tables can be accessed via the respective schema-qualified name, SYS.<table_name> (e.g. "SELECT * FROM SYS.DUAL").

There are some system tables that are critical to security, these can only be accessed by users with the "SELECT ANY DICTIONARY" system privilege (users with the DBA role have this privilege implicitly). This includes all system tables with the "EXA_DBA_" prefix.

There are also system tables to which everyone has access, however, the content of these is dependent on the current user. In EXA_ALL_OBJECTS, for example, only the database objects the current user has access to are displayed.

A.2. List of system tables

All of the system tables supported in Exasol including their access rules are shown below.

A.2.1. Catalog system tables

EXA_SYSCAT

This system table lists all existing system tables.

All users have access.

Column	Meaning
SCHEMA_NAME	Name of the system schema
OBJECT_NAME	Name of the system table
OBJECT_TYPE	Type of object: TABLE or VIEW
OBJECT_COMMENT	Comment on the object

A.2.2. Metadata system tables

EXA_ALL_COLUMNS

This system table contains information on all the table columns to which the current user has access.

All users have access.

Column	Meaning
COLUMN_SCHEMA	Associated schema
COLUMN_TABLE	Associated table
COLUMN_OBJECT_TYPE	Associated object type

Column	Meaning
COLUMN_NAME	Name of column
COLUMN_TYPE	Data type of column
COLUMN_TYPE_ID	ID of data type
COLUMN_MAXSIZE	Maximum number of characters for strings
COLUMN_NUM_PREC	Precision for numeric values
COLUMN_NUM_SCALE	Scale for numeric values
COLUMN_ORDINAL_POSITION	Position of the column in the table beginning at 1
COLUMN_IS_VIRTUAL	States whether the column is part of a virtual table
COLUMN_IS_NULLABLE	States whether NULL values are allowed (TRUE or FALSE). In case of views this value is always NULL.
COLUMN_IS_DISTRIBUTION_KEY	States whether the column is part of the distribution key (TRUE or FALSE)
COLUMN_DEFAULT	Default value of the column
COLUMN_IDENTITY	Current value of the identity number generator, if this column has the identity attribute.
COLUMN_OWNER	Owner of the corresponding object
COLUMN_OBJECT_ID	ID of the column
STATUS	Status of the object
COLUMN_COMMENT	Comment on the column

EXA_ALL_CONNECTIONS

Lists all connections of the database.

All users have access.

Column	Meaning
CONNECTION_NAME	Name of the connection
CREATED	Time of the creation date
CONNECTION_COMMENT	Comment on the connection

EXA_ALL_CONSTRAINTS

This system table contains information about constraints of tables to which the current user has access.

All users have access.

Column	Meaning
CONSTRAINT_SCHEMA	Associated schema
CONSTRAINT_TABLE	Associated table
CONSTRAINT_TYPE	Constraint type (PRIMARY KEY, FOREIGN KEY or NOT NULL)
CONSTRAINT_NAME	Name of the constraints
CONSTRAINT_ENABLED	Displays whether the constraint is checked or not
CONSTRAINT_OWNER	Owner of the constraint (which is the owner of the table)

EXA_ALL_CONSTRAINT_COLUMNS

This system table contains information about referenced table columns of all constraints to which the current user has access.

All users have access.

Column	Meaning
CONSTRAINT_SCHEMA	Associated schema
CONSTRAINT_TABLE	Associated table
CONSTRAINT_TYPE	Constraint type (PRIMARY KEY, FOREIGN KEY or NOT NULL)
CONSTRAINT_NAME	Name of the constraints
CONSTRAINT_OWNER	Owner of the constraint (which is the owner of the table)
ORDINAL_POSITION	Position of the column in the table beginning at 1
COLUMN_NAME	Name of the column
REFERENCED_SCHEMA	Referenced schema (only for foreign keys)
REFERENCED_TABLE	Referenced table (only for foreign keys)
REFERENCED_COLUMN	Name of the column in the referenced table (only for foreign keys)

EXA_ALL_DEPENDENCIES

Lists all direct dependencies between schema objects to which the current user has access. Please note that e.g. dependencies between scripts cannot be determined. In case of a view, entries with REFERENCE_TYPE=NULL values can be shown if underlying objects have been changed and the view has not been accessed again yet.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema of object
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
OBJECT_OWNER	Owner of the object
OBJECT_ID	ID of the object
REFERENCE_TYPE	Reference type (VIEW, CONSTRAINT)
REFERENCED_OBJECT_SCHEMA	Schema of the referenced object
REFERENCED_OBJECT_NAME	Name of the referenced object
REFERENCED_OBJECT_TYPE	Type of the referenced object
REFERENCED_OBJECT_OWNER	Owner of the referenced object
REFERENCED_OBJECT_ID	ID of the referenced object

EXA_ALL_FUNCTIONS

This system table describes all functions of the database to which the current user has access.

All users have access.

Column	Meaning
FUNCTION_SCHEMA	Schema of the function
FUNCTION_NAME	Name of the function
FUNCTION_OWNER	Owner of the function
FUNCTION_OBJECT_ID	ID of the function
FUNCTION_TEXT	Generation text of the function
FUNCTION_COMMENT	Comment on the function

EXA_ALL_INDICES

This system table describes all indices on tables to which the current user has access. Please note that indices are created and managed automatically by the system. Hence, the purpose of this table is mainly for transparency.

All users have access.

Column	Meaning
INDEX_SCHEMA	Schema of the index
INDEX_TABLE	Table of the index
INDEX_OWNER	Owner of the index
INDEX_OBJECT_ID	ID of the index
INDEX_TYPE	Index type
MEM_OBJECT_SIZE	Index size in bytes (at last COMMIT)
CREATED	Timestamp of when the index was created
LAST_COMMIT	Last time the object was changed in the DB
REMARKS	Additional information about the index

EXA_ALL_OBJ_PRIVS

This table contains all of the object privileges granted for objects in the database to which the current user has access.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object
OBJECT_TYPE	Object type
PRIVILEGE	The granted right
GRANTEE	Recipient of the right
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_ALL_OBJ_PRIVS_MADE

Lists all object privileges that are self-assigned by the user or those on objects that belong to the user.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object
OBJECT_TYPE	Object type
PRIVILEGE	The granted right
GRANTEE	Recipient of the right
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_ALL_OBJ_PRIVS_REC'D

Lists all object privileges granted to the user directly or via PUBLIC.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object
OBJECT_TYPE	Object type
PRIVILEGE	The granted right
GRANTEE	Recipient of the right
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_ALL_OBJECTS

This system table describes all of the database objects to which the current user has access.

All users have access.

Column	Meaning
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
CREATED	Timestamp of when the object was created
LAST_COMMIT	Last time the object was changed in the DB
OWNER	Owner of the object
OBJECT_ID	ID of the object
ROOT_NAME	Name of the containing object
ROOT_TYPE	Type of the containing object
ROOT_ID	ID of the containing object
OBJECT_IS_VIRTUAL	States whether this is a virtual object
OBJECT_COMMENT	Comment on the object

EXA_ALL_OBJECT_SIZES

This system table contains the sizes of all of the database objects to which the current user has access. The values are calculated recursively, i.e. the size of a schema includes the total of all of the sizes of the schema objects contained therein.

All users have access.

Column	Meaning
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
RAW_OBJECT_SIZE	Uncompressed volume of data in the object in bytes (at last COMMIT)
MEM_OBJECT_SIZE	Compressed volume of data in the object in bytes (at last COMMIT)
CREATED	Timestamp of when the object was created
LAST_COMMIT	Last time the object was changed in the DB
OWNER	Owner of the object
OBJECT_ID	ID of the object
OBJECT_IS_VIRTUAL	States whether this is a virtual object
ROOT_NAME	Name of the containing object
ROOT_TYPE	Type of the containing object
ROOT_ID	ID of the containing object

EXA_ALL_ROLES

A list of all roles known to the system

All users have access.

Column	Meaning
ROLE_NAME	Name of role
CREATED	Timestamp of when the role was created
ROLE_PRIORITY	Priority of the role
ROLE_COMMENT	Comment on the role

EXA_ALL_SCRIPTS

This system table describes all scripts of the database to which the current user has access.

All users have access.

Column	Meaning
SCRIPT_SCHEMA	Name of the schema of the script
SCRIPT_NAME	Name of the script
SCRIPT_OWNER	Owner of the script
SCRIPT_OBJECT_ID	ID of the script object
SCRIPT_TYPE	Type of the script (PROCEDURE, ADAPTER or UDF)

Column	Meaning
SCRIPT_LANGUAGE	Script language
SCRIPT_INPUT_TYPE	Script input type (NULL, SCALAR or SET)
SCRIPT_RESULT_TYPE	Return type of the script (ROWCOUNT, TABLE, RETURNS or EMITS)
SCRIPT_TEXT	Complete creation text for a script
SCRIPT_COMMENT	Comment on the script

EXA_ALL_SESSIONS

This system table contains information on user sessions. Among other things the most recent SQL statement is shown. With regard to security issues, only the command name is displayed. The complete SQL text can be found in [EXA_USER_SESSIONS](#) and [EXA_DB_SESSIONS](#).

All users have access.

Column	Meaning
SESSION_ID	Id of the session
USER_NAME	Logged-in user
STATUS	Current status of the session. The most important of these are: IDLE Client is connected but not taking any action. EXECUTE SQL Execution of one single SQL statement. EXECUTE BATCH Execution of a series of SQL statements. PREPARE SQL Entry of a prepared statement. EXECUTE PREPARED Execution of a prepared statement. FETCH DATA Client reads results. QUEUED Client is queued because too many parallel queries are executed.
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)
STMT_ID	Serinally numbered id of statement within a session
DURATION	Duration of the statement
QUERY_TIMEOUT	Session-specific value of QUERY_TIMEOUT, see also ALTER SESSION and ALTER SYSTEM
ACTIVITY	Current activity of the SQL
TEMP_DB_RAM	Current temporary database memory usage in MiB (cluster wide)
LOGIN_TIME	Time of login
CLIENT	Client application used by the user
DRIVER	Used driver
ENCRYPTED	Flag whether the connection is encrypted
PRIORITY	Priority group
NICE	NICE attribute
RESOURCES	Allocated resources in percent

EXA_ALL_TABLES

This system table describes all of the tables in the database to which the current user has access.

All users have access.

Column	Meaning
TABLE_SCHEMA	Name of the schema of the table
TABLE_NAME	Name of the table
TABLE_OWNER	Owner of the table
TABLE_OBJECT_ID	ID of the table
TABLE_IS_VIRTUAL	States whether this is a virtual table
TABLE_HAS_DISTRIBUTION_KEY	States whether the table is explicitly distributed
TABLE_ROW_COUNT	Number of rows in the table
DELETE_PERCENTAGE	Fraction of the rows which are just marked as deleted, but not yet physically deleted (in percent)
TABLE_COMMENT	Comment on the table

EXA_ALL_USERS

This table provides restricted information on all of the users known to the system.

All users have access.

Column	Meaning
USER_NAME	Name of the user
CREATED	Time the user was created
USER_PRIORITY	Priority of the user
USER_COMMENT	Comment on the user

EXA_ALL_VIEWS

Lists all of the views accessible to the current user.

All users have access.

Column	Meaning
VIEW_SCHEMA	Name of the schema in which the view was created
VIEW_NAME	Name of the view
SCOPE_SCHEMA	Schema from which the view was created
VIEW_OWNER	Owner of the view
VIEW_OBJECT_ID	Internal ID of the view
VIEW_TEXT	Text of the view, with which it was created
VIEW_COMMENT	Comment on the view

EXA_ALL_VIRTUAL_COLUMNS

Lists all columns of virtual tables to which the current user has access. It contains the information which are specific to virtual columns. Virtual columns are also listed in the table [EXA_ALL_COLUMNS](#).

All users have access.

Column	Meaning
COLUMN_SCHEMA	Associated virtual schema
COLUMN_TABLE	Associated virtual table
COLUMN_NAME	Name of the virtual column
COLUMN_OBJECT_ID	ID of the virtual columns object
ADAPTER_NOTES	The adapter can store additional information about the virtual column in this field

EXA_ALL_VIRTUAL_SCHEMA_PROPERTIES

This system table contains information on the properties of all virtual schemas to which the current user has access.

All users have access.

Column	Meaning
SCHEMA_NAME	Name of the virtual schema
SCHEMA_OBJECT_ID	ID of the virtual schema object
PROPERTY_NAME	Name of the property of the virtual schema
PROPERTY_VALUE	Value of the property of the virtual schema

EXA_ALL_VIRTUAL_TABLES

Lists all virtual tables to which the current user has access. It contains the information which are specific to virtual tables. Virtual tables are also listed in the table [EXA_ALL_TABLES](#).

All users have access.

Column	Meaning
TABLE_SCHEMA	Name of the virtual schema containing the virtual table
TABLE_NAME	Name of the virtual table
TABLE_OBJECT_ID	ID of the virtual table
LAST_REFRESH	Timestamp of the last metadata refresh (when metadata was committed)
LAST_REFRESH_BY	Name of the user that performed the last metadata refresh
ADAPTER_NOTES	The adapter can store additional information about the table in this field

EXA_DBA_COLUMNS

This system table contains information on all table columns.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
COLUMN_SCHEMA	Associated schema
COLUMN_TABLE	Associated table
COLUMN_OBJECT_TYPE	Associated object type
COLUMN_NAME	Name of column

Column	Meaning
COLUMN_TYPE	Data type of column
COLUMN_TYPE_ID	ID of data type
COLUMN_MAXSIZE	Maximum number of characters for strings
COLUMN_NUM_PREC	Precision for numeric values
COLUMN_NUM_SCALE	Scale for numeric values
COLUMN_ORDINAL_POSITION	Position of the column in the table beginning at 1
COLUMN_IS_VIRTUAL	States whether the column is part of a virtual table
COLUMN_IS_NULLABLE	States whether NULL values are allowed (TRUE or FALSE). In case of views this value is always NULL.
COLUMN_IS_DISTRIBUTION_KEY	States whether the column is part of the distribution key (TRUE or FALSE)
COLUMN_DEFAULT	Default value of the column
COLUMN_IDENTITY	Current value of the identity number generator, if this column has the identity attribute.
COLUMN_OWNER	Owner of the associated object
COLUMN_OBJECT_ID	ID of the column
STATUS	Status of the object
COLUMN_COMMENT	Comment on the column

EXA_DBA_CONNECTIONS

Lists all connections of the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
CONNECTION_NAME	Name of the connection
CONNECTION_STRING	Defines the target of the connection
USER_NAME	User name which is used when a connection is used
CREATED	Time of the creation date
CONNECTION_COMMENT	Comment on the connection

EXA_DBA_CONNECTION_PRIVS

Lists all connections which were granted to the user or one of his roles.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
GRANTEE	Name of the user/role who/which has been granted the right, to adopt GRANTED_ROLE
GRANTED_CONNECTION	Name of the connection
ADMIN_OPTION	Specifies whether GRANTEE is allowed to grant the right to the connection to other users or roles

EXA_DBA_RESTRICTED_OBJ_PRIVS

Lists all connection objects to which certain, restricted scripts were granted access to.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_SCHEMA	Schema of the object for which the restricted privilege has been granted
OBJECT_NAME	Name of the object for which the restricted privilege has been granted
OBJECT_TYPE	Type of the object for which the restricted privilege has been granted
FOR_OBJECT_SCHEMA	Schema of the object that the privilege is restricted to
FOR_OBJECT_NAME	Name of the object that the privilege is restricted to
FOR_OBJECT_TYPE	Type of the object that the privilege is restricted to
PRIVILEGE	The restricted privilege that is granted
GRANTEE	Name of the user/role who/which has been granted the restricted privilege
GRANTOR	Name of the user/role who/which has granted the restricted privilege
OWNER	Name of the owner of the object for which the restricted privilege is granted

EXA_DBA_CONSTRAINTS

This system table contains information about all constraints of the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
CONSTRAINT_SCHEMA	Associated schema
CONSTRAINT_TABLE	Associated table
CONSTRAINT_TYPE	Constraint type (PRIMARY KEY, FOREIGN KEY or NOT NULL)
CONSTRAINT_NAME	Name of the constraints
CONSTRAINT_ENABLED	Displays whether the constraint is checked or not
CONSTRAINT_OWNER	Owner of the constraint (which is the owner of the table)

EXA_DBA_CONSTRAINT_COLUMNS

This system table contains information about referenced table columns of all constraints of the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
CONSTRAINT_SCHEMA	Associated schema
CONSTRAINT_TABLE	Associated table
CONSTRAINT_TYPE	Constraint type (PRIMARY KEY, FOREIGN KEY or NOT NULL)
CONSTRAINT_NAME	Name of the constraints
CONSTRAINT_OWNER	Owner of the constraint (which is the owner of the table)
ORDINAL_POSITION	Position of the column in the table beginning at 1

Column	Meaning
COLUMN_NAME	Name of the column
REFERENCED_SCHEMA	Referenced schema (only for foreign keys)
REFERENCED_TABLE	Referenced table (only for foreign keys)
REFERENCED_COLUMN	Name of the column in the referenced table (only for foreign keys)

EXA_DBA_DEPENDENCIES

Lists all direct dependencies between schema objects. Please note that e.g. dependencies between scripts cannot be determined. In case of a view, entries with REFERENCE_TYPE=NULL values can be shown if underlying objects have been changed and the view has not been accessed again yet.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_SCHEMA	Schema of object
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
OBJECT_OWNER	Owner of the object
OBJECT_ID	ID of the object
REFERENCE_TYPE	Reference type (VIEW, CONSTRAINT)
REFERENCED_OBJECT_SCHEMA	Schema of the referenced object
REFERENCED_OBJECT_NAME	Name of the referenced object
REFERENCED_OBJECT_TYPE	Type of the referenced object
REFERENCED_OBJECT_OWNER	Owner of the referenced object
REFERENCED_OBJECT_ID	ID of the referenced object

EXA_DBA_DEPENDENCIES_RECURSIVE

Lists all direct and indirect dependencies between schema objects (thus recursive). Please note that e.g. dependencies between scripts cannot be determined. Views are not shown if underlying objects have been changed and the view has not been accessed again yet.



Please note that all database objects are read when accessing this system table.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_SCHEMA	Schema of object
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
OBJECT_OWNER	Owner of the object
OBJECT_ID	ID of the object
REFERENCE_TYPE	Reference type (VIEW, CONSTRAINT)
REFERENCED_OBJECT_SCHEMA	Schema of the referenced object
REFERENCED_OBJECT_NAME	Name of the referenced object

Column	Meaning
REFERENCED_OBJECT_TYPE	Type of the referenced object
REFERENCED_OBJECT_OWNER	Owner of the referenced object
REFERENCED_OBJECT_ID	ID of the referenced object
DEPENDENCY_LEVEL	Hierarchy level in the dependency graph

EXA_DBA_FUNCTIONS

This system table describes all function of the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
FUNCTION_SCHEMA	Schema of the function
FUNCTION_NAME	Name of the function
FUNCTION_OWNER	Owner of the function
FUNCTION_OBJECT_ID	ID of the function
FUNCTION_TEXT	Generation text of the function
FUNCTION_COMMENT	Comment on the function

EXA_DBA_INDICES

This system table describes all indices on tables. Please note that indices are created and managed automatically by the system. Hence, the purpose of this table is mainly for transparency.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
INDEX_SCHEMA	Schema of the index
INDEX_TABLE	Table of the index
INDEX_OWNER	Owner of the index
INDEX_OBJECT_ID	ID of the index
INDEX_TYPE	Index type
MEM_OBJECT_SIZE	Index size in bytes (at last COMMIT)
CREATED	Timestamp of when the index was created
LAST_COMMIT	Last time the object was changed in the DB
REMARKS	Additional information about the index

EXA_DBA_OBJ_PRIVS

This table contains all of the object privileges granted for objects in the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object

Column	Meaning
OBJECT_TYPE	Object type
PRIVILEGE	The granted right
GRANTEE	Recipient of the right
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_DBA_OBJECTS

This system table describes all of the database objects.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
CREATED	Timestamp of when the object was created
LAST_COMMIT	Last time the object was changed in the DB
OWNER	Owner of the object
OBJECT_ID	Unique ID of the object
ROOT_NAME	Name of the containing object
ROOT_TYPE	Type of the containing object
ROOT_ID	ID of the containing object
OBJECT_IS_VIRTUAL	States whether this is a virtual object
OBJECT_COMMENT	Comment on the object

EXA_DBA_OBJECT_SIZES

This system table contains the sizes of all database objects. The values are calculated recursively, i.e. the size of a schema includes the total of all of the sizes of the schema objects contained therein.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
RAW_OBJECT_SIZE	Uncompressed volume of data in the object in bytes (at last COMMIT)
MEM_OBJECT_SIZE	Compressed volume of data in the object in bytes (at last COMMIT)
CREATED	Timestamp of when the object was created
LAST_COMMIT	Last time the object was changed in the DB
OWNER	Owner of the object
OBJECT_ID	ID of the object
OBJECT_IS_VIRTUAL	States whether this is a virtual object
ROOT_NAME	Name of the containing object
ROOT_TYPE	Type of the containing object

Column	Meaning
ROOT_ID	ID of the containing object

EXA_DBA_ROLES

A list of all roles known to the system

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
ROLE_NAME	Name of role
CREATED	Timestamp of when the role was created
ROLE_PRIORITY	Priority of the role
ROLE_COMMENT	Comment on the role

EXA_DBA_ROLE_PRIVS

List of all roles granted to a user or a role.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
GRANTEE	Name of user or role who/which has been granted the right, to adopt GRANTED_ROLE
GRANTED_ROLE	Name of role that was granted
ADMIN_OPTION	Specifies whether GRANTEE is allowed to grant the right to the role to other users or roles

EXA_DBA_SCRIPTS

This system table describes all scripts of the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
SCRIPT_SCHEMA	Name of the schema of the script
SCRIPT_NAME	Name of the script
SCRIPT_OWNER	Owner of the script
SCRIPT_OBJECT_ID	ID of the script object
SCRIPT_TYPE	Type of the script (PROCEDURE, ADAPTER or UDF)
SCRIPT_LANGUAGE	Script language
SCRIPT_INPUT_TYPE	Script input type (NULL, SCALAR or SET)
SCRIPT_RESULT_TYPE	Return type of the script (ROWCOUNT, TABLE, RETURNS or EMITS)
SCRIPT_TEXT	Complete creation text for a script
SCRIPT_COMMENT	Comment on the script

EXA_DBA_SESSIONS

This system table contains information on user sessions. Among other things the most recent SQL statement is shown.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
SESSION_ID	Id of the session
USER_NAME	Logged-in user
STATUS	Current status of the session. The most important of these are: IDLE Client is connected but not taking any action. EXECUTE SQL Execution of one single SQL statement. EXECUTE BATCH Execution of a series of SQL statements. PREPARE SQL Entry of a prepared statement. EXECUTE PREPARED Execution of a prepared statement. FETCH DATA Client reads results. QUEUED Client is queued because too many parallel queries are executed.
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)
STMT_ID	Seriously numbered id of statement within a session
DURATION	Duration of the statement
QUERY_TIMEOUT	Session-specific value of QUERY_TIMEOUT, see also ALTER SESSION and ALTER SYSTEM
ACTIVITY	Current activity of the SQL
TEMP_DB_RAM	Current temporary database memory usage in MiB (cluster wide)
LOGIN_TIME	Time of login
CLIENT	Client application used by the user
DRIVER	Used driver
ENCRYPTED	Flag whether the connection is encrypted
HOST	Computer name or IP address from which the user has logged-in
OS_USER	User name under which the user logged into the operating system of the computer from which the login came
OS_NAME	Operating system of the client server
SCOPE_SCHEMA	Name of the schema in which the user is located
PRIORITY	Priority group
NICE	NICE attribute
RESOURCES	Allocated resources in percent
SQL_TEXT	SQL text of the statement

EXA_DBA_SYS_PRIVS

This table shows the system privileges granted to all users and roles.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
GRANTEE	Name of user or role who/which has been granted the privilege.

Column	Meaning
PRIVILEGE	System privilege that was granted.
ADMIN_OPTION	Specifies whether GRANTEE is allowed to grant the right.

EXA_DBA_TABLES

This system table describes all tables in the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
TABLE_SCHEMA	Name of the schema of the table
TABLE_NAME	Name of the table
TABLE_OWNER	Owner of the table
TABLE_OBJECT_ID	ID of the table
TABLE_IS_VIRTUAL	States whether this is a virtual table
TABLE_HAS DISTRIBUTION KEY	States whether the table is explicitly distributed
TABLE_ROW_COUNT	Number of rows in the table
DELETE_PERCENTAGE	Fraction of the rows which are just marked as deleted, but not yet physically deleted (in percent)
TABLE_COMMENT	Comment on the table

EXA_DBA_USERS

The DBA_USERS table provides complete information on all of the users known to the system.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
USER_NAME	Name of the user
CREATED	Time of the creation date
DISTINGUISHED_NAME	For the authorization against a LDAP server
KERBEROS_PRINCIPAL	Kerberos principal
PASSWORD	Encoded hash value of the password
USER_PRIORITY	Priority of the user
USER_COMMENT	Comment on the user

EXA_DBA_VIEWS

Lists all views in the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
VIEW_SCHEMA	Name of the schema in which the view was created
VIEW_NAME	Name of the view
SCOPE_SCHEMA	Schema from which the view was created

Column	Meaning
VIEW_OWNER	Owner of the view
VIEW_OBJECT_ID	Internal ID of the view
VIEW_TEXT	Text of the view, with which it was created
VIEW_COMMENT	Comment on the view

EXA_DBA_VIRTUAL_COLUMNS

Lists all columns of virtual tables. It contains the information which are specific to virtual columns. Virtual columns are also listed in the table [EXA_DBA_COLUMNS](#).

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
COLUMN_SCHEMA	Associated virtual schema
COLUMN_TABLE	Associated virtual table
COLUMN_NAME	Name of the virtual column
COLUMN_OBJECT_ID	ID of the virtual columns object
ADAPTER_NOTES	The adapter can store additional information about the virtual column in this field

EXA_DBA_VIRTUAL_SCHEMA_PROPERTIES

This system table lists the properties of all virtual schemas in the database.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
SCHEMA_NAME	Name of the virtual schema
SCHEMA_OBJECT_ID	ID of the virtual schema object
PROPERTY_NAME	Name of the property of the virtual schema
PROPERTY_VALUE	Value of the property of the virtual schema

EXA_DBA_VIRTUAL_TABLES

Lists all virtual tables and contains the information which are specific to virtual tables. Virtual tables are also listed in the table [EXA_DBA_TABLES](#).

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
TABLE_SCHEMA	Name of the virtual schema containing the virtual table
TABLE_NAME	Name of the virtual table
TABLE_OBJECT_ID	ID of the virtual table
LAST_REFRESH	Timestamp of the last metadata refresh (when metadata was committed)
LAST_REFRESH_BY	Name of the user that performed the last metadata refresh

Column	Meaning
ADAPTER_NOTES	The adapter can store additional information about the table in this field

EXA_LOADAVG

This system table contains information on the current CPU load in each of the Exasol nodes.

All users have access.

Column	Meaning
IPROC	Number of the node
LAST_1MIN	Average load in the last minute
LAST_5MIN	Average load in the last 5 minutes
LAST_15MIN	Average load in the last 15 minutes
RUNNING	Contains two numbers separated with "/". The first indicates the number of active processes or threads at the time of evaluation. The second indicates the overall number of processes and threads on the node.

EXA_METADATA

This system table contains information that describes the properties of the database.

All users have access.

Column	Meaning
PARAM_NAME	Name of the property
PARAM_VALUE	Value of the property
IS_STATIC	TRUE The value of this property remains constant FALSE The value of this property can change at any time

EXA_PARAMETERS

This system table provides the database parameters - both system-wide and session-based information is displayed.

All users have access.

Column	Meaning
PARAMETER_NAME	Name of the parameter
SESSION_VALUE	Value of the session parameter
SYSTEM_VALUE	Value of the system parameter

EXA_ROLE_CONNECTION_PRIVS

Lists any connection that the current user possesses indirectly via other roles.

All users have access.

Column	Meaning
GRANTEE	Name of the role which received the right
GRANTED_CONNECTION	Name of the connection which was granted
ADMIN_OPTION	Information on whether the connection can be passed on to other users/roles

EXA_ROLE_RESTRICTED_OBJ_PRIVS

Lists all connection objects to which certain, restricted scripts were granted access to, either granted directly to the current user or indirectly via other roles.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_SCHEMA	Schema of the object for which the restricted privilege has been granted
OBJECT_NAME	Name of the object for which the restricted privilege has been granted
OBJECT_TYPE	Type of the object for which the restricted privilege has been granted
FOR_OBJECT_SCHEMA	Schema of the object that the privilege is restricted to
FOR_OBJECT_NAME	Name of the object that the privilege is restricted to
FOR_OBJECT_TYPE	Type of the object that the privilege is restricted to
PRIVILEGE	The restricted privilege that is granted
GRANTEE	Name of the user/role who/which has been granted the restricted privilege
GRANTOR	Name of the user/role who/which has granted the restricted privilege
OWNER	Name of the owner of the object for which the restricted privilege is granted

EXA_ROLE_OBJ_PRIVS

Lists all object privileges that have been granted to roles of the user.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object
OBJECT_TYPE	Object type
PRIVILEGE	The granted right
GRANTEE	Name of the role
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_ROLE_ROLE_PRIVS

Lists any role that the current user possesses indirectly via other roles.

All users have access.

Column	Meaning
GRANTEE	Name of the role through which the user possesses the role indirectly
GRANTED_ROLE	Name of the role possessed by the user indirectly
ADMIN_OPTION	Information on whether the role can be passed on to other users/roles

EXA_ROLE_SYS_PRIVS

Lists any system privilege that the current user possesses via roles.

All users have access.

Column	Meaning
GRANTEE	Name of the role
PRIVILEGE	Name of the system privilege
ADMIN_OPTION	Shows whether it is permissible for the privilege to be passed on to other users/roles

EXA_SCHEMAS

This system table lists all the schemas of the database.

All users have access.

Column	Meaning
SCHEMA_NAME	Name of the schema object
SCHEMA_OWNER	Owner of the object
SCHEMA_OBJECT_ID	ID of the schema object
SCHEMA_IS_VIRTUAL	States whether this is a virtual schema
SCHEMA_COMMENT	Comment on the object

EXA_SCHEMA_OBJECTS

Lists all objects that exist in the current schema.

All users have access.

Column	Meaning
OBJECT_NAME	Name of the object
OBJECT_TYPE	Type of the object

EXA_SESSION_CONNECTIONS

List of all connections the user can access. The entries for columns CONNECTION_STRING and USER_NAME are only displayed if the user can edit the connection (see also [ALTER CONNECTION](#)).

All users have access.

Column	Meaning
CONNECTION_NAME	Name of the connection

Column	Meaning
CONNECTION_STRING	Defines the target of the connection
USER_NAME	User name which is used when a connection is used
CREATED	Time of the creation date
CONNECTION_COMMENT	Comment on the connection

EXA_SESSION_PRIVS

Lists all of the system privileges the user currently disposes of.

All users have access.

Column	Meaning
PRIVILEGE	Name of the system privilege

EXA_SESSION_ROLES

Lists all roles held by the current user.

All users have access.

Column	Meaning
ROLE_NAME	Name of the role
ROLE_PRIORITY	Priority of the role
ROLE_COMMENT	Comment on the role

EXA_SPATIAL_REF_SYS

List of supported spatial reference systems.

All users have access.

Column	Meaning
SRID	Spatial reference system identifier
AUTH_NAME	Spatial reference system authority name
AUTH_SRID	Authority specific spatial reference system identifier
SRTEXT	WKT description of the spatial reference system
PROJ4TEXT	Parameters for Proj4 projections

EXA_SQL_KEYWORDS

This system table contains all SQL keywords in Exasol.

All users have access.

Column	Meaning
KEYWORD	Keyword
RESERVED	Defines whether the keyword is reserved. Reserved keywords cannot be used as SQL identifier (see also Section 2.1.2, “SQL identifier”).

EXA_SQL_TYPES

This system table describes the SQL data types of Exasol.

All users have access.

Column	Meaning
TYPE_NAME	Name according to SQL standard
TYPE_ID	ID of the data type
PRECISION	The precision in relation to numeric values, the (maximum) length in bytes in relation to strings and other types.
LITERAL_PREFIX	Prefix, with which a literal of this type must be initiated
LITERAL_SUFFIX	Suffix, with which a literal of this type must be terminated
CREATE_PARAMS	Information on which information is necessary in order to create a column of this type
IS_NULLABLE	States whether NULL values are allowed (TRUE or FALSE).
CASE_SENSITIVE	States whether case sensitivity is relevant to the type
SEARCHABLE	States how the type can be used in a WHERE clause: 0 Cannot be searched 1 Can only be searched with WHERE .. LIKE 2 Cannot be searched with WHERE .. LIKE 3 Can be searched with any WHERE clause
UNSIGNED_ATTRIBUTE	States whether the type is unsigned
FIXED_PREC_SCALE	States whether the type has fixed representation
AUTO_INCREMENT	States whether it is an automatically incremented type
LOCAL_TYPE_NAME	Local type name
MINIMUM_SCALE	Minimum number of digits after the decimal point
MAXIMUM_SCALE	Maximum number of digits after the decimal point
SQL_DATA_TYPE	Reserved
SQL_DATETIME_SUB	Reserved
NUM_PREC_RADIX	Number base for details of the precision
INTERVAL_PRECISION	Reserved

EXA_STATISTICS_OBJECT_SIZES

This system table contains the sizes of all statistical system tables aggregated by the type of the statistical info (see also [Section A.2.3, “Statistical system tables”](#)).

All users have access.

Column	Meaning
STATISTICS_TYPE	Type of statistics
	AUDIT Auditing information
	DB_SIZE Statistics about the size of database objects
	MONITOR All monitoring statistics

Column	Meaning
	OTHER Miscellaneous statistics, e.g. for internal optimizations
	PROFILE Profiling information
	SQL Statistics about SQL statements
	SYSTEM_EVENTS Statistics about system events
	TRANSACTION_CONFLICTS Statistics about transaction conflicts
	USAGE Statistics about the usage of the system
RAW_OBJECT_SIZE	Uncompressed data volume in bytes
MEM_OBJECT_SIZE	Compressed data volume in bytes
AUXILIARY_SIZE	Auxiliary structures like indices in bytes

EXA_TIME_ZONES

This system table lists all named timezones supported by the database.

All users have access.

Column	Meaning
TIME_ZONE_NAME	Time zone name

EXA_USER_COLUMNS

This system table contains information on the table columns to those tables owned by the current user.

All users have access.

Column	Meaning
COLUMN_SCHEMA	Associated schema
COLUMN_TABLE	Associated table
COLUMN_OBJECT_TYPE	Associated object type
COLUMN_NAME	Name of column
COLUMN_TYPE	Data type of column
COLUMN_TYPE_ID	ID of data type
COLUMN_MAXSIZE	Maximum number of characters for strings
COLUMN_NUM_PREC	Precision for numeric values
COLUMN_NUM_SCALE	Scale for numeric values
COLUMN_ORDINAL_POSITION	Position of the column in the table beginning at 1
COLUMN_IS_VIRTUAL	States whether the column is part of a virtual table
COLUMN_IS_NULLABLE	States whether NULL values are allowed (TRUE or FALSE). In case of views this value is always NULL.
COLUMN_IS_DISTRIBUTION_KEY	States whether the column is part of the distribution key (TRUE or FALSE)
COLUMN_DEFAULT	Default value of the column

Column	Meaning
COLUMN_IDENTITY	Current value of the identity number generator, if this column has the identity attribute.
COLUMN_OWNER	Owner of the associated object
COLUMN_OBJECT_ID	ID of the column
STATUS	Status of the object
COLUMN_COMMENT	Comment on the column

EXA_USER_CONNECTION_PRIVS

Lists all connections which were granted directly to the user.

All users have access.

Column	Meaning
GRANTEE	Name of the user who received the right
GRANTED_CONNECTION	Name of the connection which was granted
ADMIN_OPTION	Information on whether the connection can be passed on to other users/roles

EXA_USER_RESTRICTED_OBJ_PRIVS

Lists all connection objects to which certain, restricted scripts were granted access to, granted directly to the current user.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
OBJECT_SCHEMA	Schema of the object for which the restricted privilege has been granted
OBJECT_NAME	Name of the object for which the restricted privilege has been granted
OBJECT_TYPE	Type of the object for which the restricted privilege has been granted
FOR_OBJECT_SCHEMA	Schema of the object that the privilege is restricted to
FOR_OBJECT_NAME	Name of the object that the privilege is restricted to
FOR_OBJECT_TYPE	Type of the object that the privilege is restricted to
PRIVILEGE	The restricted privilege that is granted
GRANTEE	Name of the user/role who/which has been granted the restricted privilege
GRANTOR	Name of the user/role who/which has granted the restricted privilege
OWNER	Name of the owner of the object for which the restricted privilege is granted

EXA_USER_CONSTRAINTS

This system table contains information about constraints of tables owned by the current user.

All users have access.

Column	Meaning
CONSTRAINT_SCHEMA	Associated schema

Column	Meaning
CONSTRAINT_TABLE	Associated table
CONSTRAINT_TYPE	Constraint type (PRIMARY KEY, FOREIGN KEY or NOT NULL)
CONSTRAINT_NAME	Name of the constraints
CONSTRAINT_ENABLED	Displays whether the constraint is checked or not
CONSTRAINT_OWNER	Owner of the constraint (which is the owner of the table)

EXA_USER_CONSTRAINT_COLUMNS

This system table contains information about referenced table columns of all constraints owned by the current user.

All users have access.

Column	Meaning
CONSTRAINT_SCHEMA	Associated schema
CONSTRAINT_TABLE	Associated table
CONSTRAINT_TYPE	Constraint type (PRIMARY KEY, FOREIGN KEY or NOT NULL)
CONSTRAINT_NAME	Name of the constraints
CONSTRAINT_OWNER	Owner of the constraint (which is the owner of the table)
ORDINAL_POSITION	Position of the column in the table beginning at 1
COLUMN_NAME	Name of the column
REFERENCED_SCHEMA	Referenced schema (only for foreign keys)
REFERENCED_TABLE	Referenced table (only for foreign keys)
REFERENCED_COLUMN	Name of the column in the referenced table (only for foreign keys)

EXA_USER_DEPENDENCIES

Lists all direct dependencies between schema objects owned by the current user. Please note that e.g. dependencies between scripts cannot be determined. In case of a view, entries with REFERENCE_TYPE=NULL values can be shown if underlying objects have been changed and the view has not been accessed again yet.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema of object
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
OBJECT_OWNER	Owner of the object
OBJECT_ID	ID of the object
REFERENCE_TYPE	Reference type (VIEW, CONSTRAINT)
REFERENCED_OBJECT_SCHEMA	Schema of the referenced object
REFERENCED_OBJECT_NAME	Name of the referenced object
REFERENCED_OBJECT_TYPE	Type of the referenced object

Column	Meaning
REFERENCED_OBJECT_OWNER	Owner of the referenced object
REFERENCED_OBJECT_ID	ID of the referenced object

EXA_USER_FUNCTIONS

This system table describes all functions in the database owned by the current user.

All users have access.

Column	Meaning
FUNCTION_SCHEMA	Schema of the function
FUNCTION_NAME	Name of the function
FUNCTION_OWNER	Owner of the function
FUNCTION_OBJECT_ID	ID of the function
FUNCTION_TEXT	Generation text of the function
FUNCTION_COMMENT	Comment on the function

EXA_USER_INDICES

This system table describes all indices on tables owned by the current user. Please note that indices are created and managed automatically by the system. Hence, the purpose of this table is mainly for transparency.

All users have access.

Column	Meaning
INDEX_SCHEMA	Schema of the index
INDEX_TABLE	Table of the index
INDEX_OWNER	Owner of the index
INDEX_OBJECT_ID	ID of the index
INDEX_TYPE	Index type
MEM_OBJECT_SIZE	Index size in bytes (at last COMMIT)
CREATED	Timestamp of when the index was created
LAST_COMMIT	Last time the object was changed in the DB
REMARKS	Additional information about the index

EXA_USER_OBJ_PRIVS

This table contains all of the object privileges granted for objects in the database to which the current user has access (except via the PUBLIC role).

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object
OBJECT_TYPE	Object type
PRIVILEGE	The granted right

Column	Meaning
GRANTEE	Recipient of the right
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_USER_OBJ_PRIVS_MADE

Lists all of the object privileges related to objects owned by the current user.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object
OBJECT_TYPE	Object type
PRIVILEGE	The granted right
GRANTEE	Recipient of the right
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_USER_OBJ_PRIVS_REC'D

Lists all object privileges granted directly to the user.

All users have access.

Column	Meaning
OBJECT_SCHEMA	Schema in which the target object is located
OBJECT_NAME	Name of the object
OBJECT_TYPE	Object type
PRIVILEGE	The granted right
GRANTEE	Recipient of the right
GRANTOR	Name of the user who granted the right
OWNER	Owner of the target object

EXA_USER_OBJECTS

This system table lists all of the objects owned by the current user.

All users have access.

Column	Meaning
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
CREATED	Timestamp of when the object was created
LAST_COMMIT	Last time the object was changed in the DB
OWNER	Owner of the object

Column	Meaning
OBJECT_ID	ID of the object
ROOT_NAME	Name of the containing object
ROOT_TYPE	Type of the containing object
ROOT_ID	ID of the containing object
OBJECT_IS_VIRTUAL	States whether this is a virtual object
OBJECT_COMMENT	Comment on the object

EXA_USER_OBJECT_SIZES

Contains the size of all database objects owned by the current user. The values are calculated recursively, i.e. the size of a schema includes the total of all of the sizes of the schema objects contained therein.

All users have access.

Column	Meaning
OBJECT_NAME	Name of object
OBJECT_TYPE	Type of object
RAW_OBJECT_SIZE	Uncompressed volume of data in the object in bytes (at last COMMIT)
MEM_OBJECT_SIZE	Compressed volume of data in the object in bytes (at last COMMIT)
CREATED	Timestamp of when the object was created
LAST_COMMIT	Last time the object was changed in the DB
OWNER	Owner of the object
OBJECT_ID	ID of the object
OBJECT_IS_VIRTUAL	States whether this is a virtual object
ROOT_NAME	Name of the containing object
ROOT_TYPE	Type of the containing object
ROOT_ID	ID of the containing object

EXA_USER_ROLE_PRIVS

This table lists all of the roles directly granted to the current user (not via other roles)

All users have access.

Column	Meaning
GRANTEE	Name of the user
GRANTED_ROLE	Name of the role possessed by the user
ADMIN_OPTION	Information on whether the role can be passed on to other users/roles

EXA_USER_SCRIPTS

This system table describes all of the scripts in the database owned by the current user.

All users have access.

Column	Meaning
SCRIPT_SCHEMA	Name of the schema of the script
SCRIPT_NAME	Name of the script
SCRIPT_OWNER	Owner of the script
SCRIPT_OBJECT_ID	ID of the script object
SCRIPT_TYPE	Type of the script (PROCEDURE, ADAPTER or UDF)
SCRIPT_LANGUAGE	Script language
SCRIPT_INPUT_TYPE	Script input type (NULL, SCALAR or SET)
SCRIPT_RESULT_TYPE	Return type of the script (ROWCOUNT, TABLE, RETURNS or EMITS)
SCRIPT_TEXT	Complete creation text for a script
SCRIPT_COMMENT	Comment on the script

EXA_USER_SESSIONS

This system table contains information on user sessions. Among other things the most recent SQL statement is shown.

All users have access.

Column	Meaning
SESSION_ID	Id of the session
USER_NAME	Logged-in user
STATUS	Current status of the session. The most important of these are: IDLE Client is connected but not taking any action. EXECUTE SQL Execution of one single SQL statement. EXECUTE BATCH Execution of a series of SQL statements. PREPARE SQL Entry of a prepared statement. EXECUTE PREPARED Execution of a prepared statement. FETCH DATA Client reads results. QUEUED Client is queued because too many parallel queries are executed.
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)
STMT_ID	Seriously numbered id of statement within a session
DURATION	Duration of the statement
QUERY_TIMEOUT	Session-specific value of QUERY_TIMEOUT, see also ALTER SESSION and ALTER SYSTEM
ACTIVITY	Current activity of the SQL
TEMP_DB_RAM	Current temporary database memory usage in MiB (cluster wide)
LOGIN_TIME	Time of login
CLIENT	Client application used by the user
DRIVER	Used driver
ENCRYPTED	Flag whether the connection is encrypted
HOST	Computer name or IP address from which the user has logged-in
OS_USER	User name under which the user logged into the operating system of the computer from which the login came

Column	Meaning
OS_NAME	Operating system of the client server
SCOPE_SCHEMA	Name of the schema in which the user is located
PRIORITY	Priority group
NICE	NICE attribute
RESOURCES	Allocated resources in percent
SQL_TEXT	SQL text of the statement

EXA_USER_SYS_PRIVS

Lists all system privileges that have been directly granted to the user.

All users have access.

Column	Meaning
GRANTEE	Name of the user
PRIVILEGE	Name of the system privilege
ADMIN_OPTION	States whether it is permissible for the system privilege to be passed on to other users/roles

EXA_USER_TABLES

This system table describes all of the tables in the database owned by the current user.

All users have access.

Column	Meaning
TABLE_SCHEMA	Name of the schema of the table
TABLE_NAME	Name of the table
TABLE_OWNER	Owner of the table
TABLE_OBJECT_ID	ID of the table
TABLE_IS_VIRTUAL	States whether this is a virtual table
TABLE_HAS DISTRIBUTION KEY	States whether the table is explicitly distributed
TABLE_ROW_COUNT	Number of rows in the table
DELETE_PERCENTAGE	Fraction of the rows which are just marked as deleted, but not yet physically deleted (in percent)
TABLE_COMMENT	Comment on the table

EXA_USER_USERS

This table provides the same information as EXA_ALL_USERS, however, it is limited to the user currently logged in.

All users have access.

Column	Meaning
USER_NAME	Name of the user
CREATED	Time the user was created

Column	Meaning
USER_PRIORITY	Priority of the user
USER_COMMENT	Comment on the user

EXA_USER_VIEWS

Lists all views owned by the current user.

All users have access.

Column	Meaning
VIEW_SCHEMA	Name of the schema in which the view was created
VIEW_NAME	Name of the view
SCOPE_SCHEMA	Schema from which the view was created
VIEW_OWNER	Owner of the view
VIEW_OBJECT_ID	Internal ID of the view
VIEW_TEXT	Text of the view, with which it was created
VIEW_COMMENT	Comment on the view

EXA_USER_VIRTUAL_COLUMNS

Lists all columns of virtual tables owned by the current user. It contains the information which are specific to virtual columns. Virtual columns are also listed in the table [EXA_USER_COLUMNS](#).

All users have access.

Column	Meaning
COLUMN_SCHEMA	Associated virtual schema
COLUMN_TABLE	Associated virtual table
COLUMN_NAME	Name of the virtual column
COLUMN_OBJECT_ID	ID of the virtual columns object
ADAPTER_NOTES	The adapter can store additional information about the virtual column in this field

EXA_USER_VIRTUAL_SCHEMA_PROPERTIES

This system table contains information on the properties of all virtual schemas belonging to the current user.

All users have access.

Column	Meaning
SCHEMA_NAME	Name of the virtual schema
SCHEMA_OBJECT_ID	ID of the virtual schema object
PROPERTY_NAME	Name of the property of the virtual schema
PROPERTY_VALUE	Value of the property of the virtual schema

EXA_USER_VIRTUAL_TABLES

Lists all virtual tables owned by the current user. It contains the information which are specific to virtual tables. Virtual tables are also listed in the table [EXA_USER_TABLES](#).

All users have access.

Column	Meaning
TABLE_SCHEMA	Name of the virtual schema containing the virtual table
TABLE_NAME	Name of the virtual table
TABLE_OBJECT_ID	ID of the virtual table
LAST_REFRESH	Timestamp of the last metadata refresh (when metadata was committed)
LAST_REFRESH_BY	Name of the user that performed the last metadata refresh
ADAPTER_NOTES	The adapter can store additional information about the table in this field

EXA_VIRTUAL_SCHEMAS

Lists all virtual schemas and shows the properties which are specific to virtual schemas. Virtual schemas are also listed in the table [EXA_SCHEMAS](#).

All users have access.

Column	Meaning
SCHEMA_NAME	Name of the virtual schema
SCHEMA_OWNER	Owner of the virtual schema
SCHEMA_OBJECT_ID	ID of the virtual schema object
ADAPTER_SCRIPT	Name of the adapter script used for this virtual schema
LAST_REFRESH	Timestamp of the last metadata refresh
LAST_REFRESH_BY	Name of the user that performed the last metadata refresh
ADAPTER_NOTES	The adapter can store additional information about the schema in this field

EXA_VOLUME_USAGE

Shows details of the database usage of the storage volumes.

All users have access.

Column	Meaning
TABLESPACE	The tablespace of the volume
VOLUME_ID	The identifier of the volume
IPROC	Number of the node
LOCALITY	Describes the locality of the master segment. If the value is FALSE, then a performance degradation of I/O operations is probable.
REDUNDANCY	Redundancy level of the volume
HDD_TYPE	Type of HDDs of the volume
HDD_COUNT	Number of HDDs of the volume

Column	Meaning
HDD_FREE	Physical free space on the node in GiB of all disks of the volume
VOLUME_SIZE	Size of the volume on the node in GiB
USE	Usage of the volume on the node in percent, i.e. 100% - (UNUSED_DATA/VOLUME_SIZE)
COMMIT_DATA	Committed data on the node in GiB
SWAP_DATA	Swapped data on the node in GiB
UNUSED_DATA	Unused data on the node in GiB. Be aware that volume fragmentation might prevent complete use of all unused data.
DELETE_DATA	Deleted data on the node in GiB. This includes all data which cannot be deleted immediately, e.g. because of an active backup or shrink operation.
MULTICOPY_DATA	Multicopy data on the node in GiB. Multicopy data occurs in case of transactions where one transaction reads old data and the other one writes new data of the same table in parallel.

A.2.3. Statistical system tables

The following system tables contain historical data about the usage and the status of the DBMS. They are placed in a special system schema EXA_STATISTICS which is automatically integrated in the namespace. These system tables can also be retrieved in the EXA_SYSCAT system table. Timestamps of historical statistics are stored in the current database time zone ([DBTIMEZONE](#)).

Statistics are updated periodically, for an explicit update the command [FLUSH STATISTICS](#) is available (see [Section 2.2.6, “Other statements”](#)).

Statistical system tables can be accessed by all users read-only and are subject of the transactional concept (see [Section 3.1, “Transaction management”](#)). Therefore you maybe have to open a new transaction to see the up-to-date data!



To minimize transactional conflicts for the user and the DBMS, within a transaction you should access exclusively either statistical system tables or normal database objects.

EXA_DBA_AUDIT_SESSIONS

Lists all sessions if auditing is switched on in EXAoperation.

This system table can be cleared by the statement [TRUNCATE AUDIT LOGS](#).

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
SESSION_ID	Id of the session
LOGIN_TIME	Login time
LOGOUT_TIME	Logout time
USER_NAME	User name
CLIENT	Client application used by the user
DRIVER	Used driver
ENCRYPTED	Flag whether the connection is encrypted
HOST	Computer name or IP address from which the user has logged-in
OS_USER	User name under which the user logged into the operating system of the computer from which the login came
OS_NAME	Operating system of the client server
SUCCESS	TRUE Login was successfully FALSE Login failed (e.g. because of a wrong password)
ERROR_CODE	Error code if the login failed
ERROR_TEXT	Error text if the login failed

EXA_DBA_AUDIT_SQL

Lists all executed SQL statements if the auditing is switched on in EXAoperation.

This system table can be cleared by the statement [TRUNCATE AUDIT LOGS](#).

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning	
SESSION_ID	Id of the session (see also EXA_DB_AUDIT_SESSIONS)	
STMT_ID	Serially numbered id of statement within a session	
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)	
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)	
DURATION	Duration of the statement in seconds	
START_TIME	Start point of the statement	
STOP_TIME	Stop point of the statement	
CPU	CPU utilization in percent	
TEMP_DB_RAM_PEAK	Maximal usage of temporary DB memory of the query in MiB (cluster wide)	
HDD_READ	Hard disk read ratio in MiB per second (per node, averaged over the duration) ☞ If this value is larger than 0, then data had to be loaded into the main memory.	
HDD_WRITE	Hard disk write ratio in MiB per second (per node, averaged over the duration) ☞ This column reflects only the data written during a COMMIT. For other statements its value is NULL.	
NET	Network traffic ratio in MiB per second (sum of send/receive, per node, averaged over the duration)	
SUCCESS	TRUE Statement was executed successfully FALSE Statement failed (e.g. with a data exception)	
ERROR_CODE	Error code if the statement failed	
ERROR_TEXT	Error text if the statement failed	
SCOPE_SCHEMA	Schema where the user was located	
PRIORITY	Priority group	
NICE	NICE attribute	
RESOURCES	Allocated resources in percent	
ROW_COUNT	Number of result rows for queries, or number of affected rows for DML and DDL statements	
EXECUTION_MODE	EXECUTE Normal execution of statements PREPARE Prepared phase for prepared statements CACHED Query which accesses the Query Cache PREPROCESS Execution of the Preprocessor script	
SQL_TEXT	SQL text limited to 2,000,000 characters	

[EXA_DB_APROFILE_LAST_DAY](#)

Lists all profiling information of sessions with activated profiling. Details for this topic can also be found in [Section 3.9, “Profiling”](#).

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning										
SESSION_ID	Id of the session										
STMT_ID	Serially numbered id of statement within a session										
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)										
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)										
PART_ID	Serially numbered id of the execution part within the statement										
PART_NAME	Name of the execution part (see also Section 3.9, “Profiling”)										
PART_INFO	Extended information of the execution part <table> <tr> <td>GLOBAL</td><td>Global action (e.g. global join)</td></tr> <tr> <td>EXPRESSION INDEX</td><td>Non-persistent join index which is built on an expression</td></tr> <tr> <td>NL JOIN</td><td>Nested loop join (cross product)</td></tr> <tr> <td>REPLICATED</td><td>Replicated object (e.g. small tables)</td></tr> <tr> <td>TEMPORARY</td><td>Temporary object (for intermediate results)</td></tr> </table>	GLOBAL	Global action (e.g. global join)	EXPRESSION INDEX	Non-persistent join index which is built on an expression	NL JOIN	Nested loop join (cross product)	REPLICATED	Replicated object (e.g. small tables)	TEMPORARY	Temporary object (for intermediate results)
GLOBAL	Global action (e.g. global join)										
EXPRESSION INDEX	Non-persistent join index which is built on an expression										
NL JOIN	Nested loop join (cross product)										
REPLICATED	Replicated object (e.g. small tables)										
TEMPORARY	Temporary object (for intermediate results)										
OBJECT_SCHEMA	Schema of the processed object										
OBJECT_NAME	Name of the processed object										
OBJECT_ROWS	Number of rows of the processed object										
OUT_ROWS	Number of result rows of the execution part										
DURATION	Duration of the execution part in seconds										
CPU	CPU utilization in percent of the execution part (averaged over the duration)										
TEMP_DB_RAM_PEAK	Usage of temporary DB memory of the execution part in MiB (cluster wide, maximum over the duration)										
HDD_READ	Hard disk read ratio in MiB per second (per node, averaged over the duration) <p> If this value is larger than 0, then data had to be loaded into the main memory.</p>										
HDD_WRITE	Hard disk write ratio in MiB per second (per node, averaged over the duration) <p> This column reflects only the data written during a COMMIT. For other statements its value is NULL.</p>										
NET	Network traffic ratio in MiB per second (sum of send/receive, per node, averaged over the duration)										
REMARKS	Additional information										
SQL_TEXT	Corresponding SQL text										

EXA_DBA_PROFILE_RUNNING

Lists all profiling information of running queries. Details for this topic can also be found in [Section 3.9, “Profiling”](#).

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
SESSION_ID	Id of the session
STMT_ID	Serially numbered id of statement within a session

Column	Meaning										
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)										
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)										
PART_ID	Serially numbered id of the execution part within the statement										
PART_NAME	Name of the execution part (see also Section 3.9, “Profiling”)										
PART_INFO	Extended information of the execution part <table> <tr> <td>GLOBAL</td><td>Global action (e.g. global join)</td></tr> <tr> <td>EXPRESSION INDEX</td><td>Non-persistent join index which is built on an expression</td></tr> <tr> <td>NL JOIN</td><td>Nested loop join (cross product)</td></tr> <tr> <td>REPLICATED</td><td>Replicated object (e.g. small tables)</td></tr> <tr> <td>TEMPORARY</td><td>Temporary object (for intermediate results)</td></tr> </table>	GLOBAL	Global action (e.g. global join)	EXPRESSION INDEX	Non-persistent join index which is built on an expression	NL JOIN	Nested loop join (cross product)	REPLICATED	Replicated object (e.g. small tables)	TEMPORARY	Temporary object (for intermediate results)
GLOBAL	Global action (e.g. global join)										
EXPRESSION INDEX	Non-persistent join index which is built on an expression										
NL JOIN	Nested loop join (cross product)										
REPLICATED	Replicated object (e.g. small tables)										
TEMPORARY	Temporary object (for intermediate results)										
PART_FINISHED	Defines whether the execution part has already been finished										
OBJECT_SCHEMA	Schema of the processed object										
OBJECT_NAME	Name of the processed object										
OBJECT_ROWS	Number of rows of the processed object										
OUT_ROWS	Number of result rows of the execution part										
DURATION	Duration of the execution part in seconds										
CPU	CPU utilization in percent of the execution part (averaged over the duration)										
TEMP_DB_RAM_PEAK	Usage of temporary DB memory of the execution part in MiB (cluster wide, maximum over the duration)										
HDD_READ	Hard disk read ratio in MiB per second (per node, averaged over the duration) <p> If this value is larger than 0, then data had to be loaded into the main memory.</p>										
HDD_WRITE	Hard disk write ratio in MiB per second (per node, averaged over the duration) <p> This column reflects only the data written during a COMMIT. For other statements its value is NULL.</p>										
NET	Network traffic ratio in MiB per second (sum of send/receive, per node, averaged over the duration)										
REMARKS	Additional information										
SQL_TEXT	Corresponding SQL text										

EXA_DBA_SESSIONS_LAST_DAY

Lists all sessions of the last day.

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
SESSION_ID	Id of the session
LOGIN_TIME	Time of login
LOGOUT_TIME	Time of logout
USER_NAME	User name

Column	Meaning
CLIENT	Client application used by the user
DRIVER	Used driver
ENCRYPTED	Flag whether the connection is encrypted
HOST	Computer name or IP address from which the user has logged-in
OS_USER	User name under which the user logged into the operating system of the computer from which the login came
OS_NAME	Operating system of the client server
SUCCESS	Information whether the login was successful
ERROR_CODE	Error code if the login failed
ERROR_TEXT	Error text if the login failed

EXA_DBA_TRANSACTION_CONFLICTS

Lists all transaction conflicts.

This system table can be cleared by the statement [TRUNCATE AUDIT LOGS](#).

Only users with the "SELECT ANY DICTIONARY" system privilege have access.

Column	Meaning
SESSION_ID	Id of the session
CONFLICT_SESSION_ID	Session which produces the conflict
START_TIME	Start time of the conflict
STOP_TIME	End time of the conflict or NULL if the conflict is still open
CONFLICT_TYPE	Type of the conflict: WAIT FOR COMMIT One session has to wait until the other one is committed TRANSACTION ROLL-BACK One session has to be rolled back due to a conflict
CONFLICT_OBJECTS	Name of the corresponding objects
CONFLICT_INFO	Additional information about the conflict

EXA_DB_SIZE_LAST_DAY

This system table contains the database sizes of the recent 24 hours. The information is aggregated across all cluster nodes.

All users have access.

Column	Meaning
MEASURE_TIME	Point of the measurement
RAW_OBJECT_SIZE	Uncompressed data volume in GiB
MEM_OBJECT_SIZE	Compressed data volume in GiB
AUXILIARY_SIZE	Size in GiB of auxiliary structures like indices
STATISTICS_SIZE	Size in GiB of statistical system tables

Column	Meaning
RECOMMENDED_DB_RAM_SIZE	Recommended DB RAM size in GiB to exploit the maximal system performance
STORAGE_SIZE	Size of the persistent volume in GiB
USE	Ratio of effectively used space of the persistent volume size in percent
OBJECT_COUNT	Number of schema objects in the database.

EXA_DB_SIZE_HOURLY

This system table describes the hourly aggregated database sizes sorted by the interval start.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
RAW_OBJECT_SIZE_AVG	Average uncompressed data volume in GiB
RAW_OBJECT_SIZE_MAX	Maximum uncompressed data volume in GiB
MEM_OBJECT_SIZE_AVG	Average compressed data volume in GiB
MEM_OBJECT_SIZE_MAX	Maximum compressed data volume in GiB
AUXILIARY_SIZE_AVG	Average size in GiB of auxiliary structures like indices
AUXILIARY_SIZE_MAX	Maximum size in GiB of auxiliary structures like indices
STATISTICS_SIZE_AVG	Average size in GiB of statistical system tables
STATISTICS_SIZE_MAX	Maximum size in GiB of statistical system tables
RECOMMENDED_DB_RAM_SIZE_AVG	Average recommended DB RAM size in GiB to exploit the optimal system performance
RECOMMENDED_DB_RAM_SIZE_MAX	Maximum recommended DB RAM size in GiB to exploit the optimal system performance
STORAGE_SIZE_AVG	Average size of the persistent volume in GiB
STORAGE_SIZE_MAX	Maximum size of the persistent volume in GiB
USE_AVG	Average ratio of effectively used space of the persistent volume size in percent
USE_MAX	Maximum ratio of effectively used space of the persistent volume size in percent
OBJECT_COUNT_AVG	Average number of schema objects in the database.
OBJECT_COUNT_MAX	Maximum number of schema objects in the database.

EXA_DB_SIZE_DAILY

This system table describes the daily aggregated database sizes sorted by the interval start.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
RAW_OBJECT_SIZE_AVG	Average uncompressed data volume in GiB
RAW_OBJECT_SIZE_MAX	Maximum uncompressed data volume in GiB

Column	Meaning
MEM_OBJECT_SIZE_AVG	Average compressed data volume in GiB
MEM_OBJECT_SIZE_MAX	Maximum compressed data volume in GiB
AUXILIARY_SIZE_AVG	Average size in GiB of auxiliary structures like indices
AUXILIARY_SIZE_MAX	Maximum size in GiB of auxiliary structures like indices
STATISTICS_SIZE_AVG	Average size in GiB of statistical system tables
STATISTICS_SIZE_MAX	Maximum size in GiB of statistical system tables
RECOMMENDED_DB_RAM_SIZE_AVG	Average recommended DB RAM size in GiB to exploit the optimal system performance
RECOMMENDED_DB_RAM_SIZE_MAX	Maximum recommended DB RAM size in GiB to exploit the optimal system performance
STORAGE_SIZE_AVG	Average size of the persistent volume in GiB
STORAGE_SIZE_MAX	Maximum size of the persistent volume in GiB
USE_AVG	Average ratio of effectively used space of the persistent volume size in percent
USE_MAX	Maximum ratio of effectively used space of the persistent volume size in percent
OBJECT_COUNT_AVG	Average number of schema objects in the database.
OBJECT_COUNT_MAX	Maximum number of schema objects in the database.

EXA_DB_SIZE_MONTHLY

This system table describes the monthly aggregated database sizes sorted by the interval start.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
RAW_OBJECT_SIZE_AVG	Average uncompressed data volume in GiB
RAW_OBJECT_SIZE_MAX	Maximum uncompressed data volume in GiB
MEM_OBJECT_SIZE_AVG	Average compressed data volume in GiB
MEM_OBJECT_SIZE_MAX	Maximum compressed data volume in GiB
AUXILIARY_SIZE_AVG	Average size in GiB of auxiliary structures like indices
AUXILIARY_SIZE_MAX	Maximum size in GiB of auxiliary structures like indices
STATISTICS_SIZE_AVG	Average size in GiB of statistical system tables
STATISTICS_SIZE_MAX	Maximum size in GiB of statistical system tables
RECOMMENDED_DB_RAM_SIZE_AVG	Average recommended DB RAM size in GiB to exploit the optimal system performance
RECOMMENDED_DB_RAM_SIZE_MAX	Maximum recommended DB RAM size in GiB to exploit the optimal system performance
STORAGE_SIZE_AVG	Average size of the persistent volume in GiB
STORAGE_SIZE_MAX	Maximum size of the persistent volume in GiB
USE_AVG	Average ratio of effectively used space of the persistent volume size in percent
USE_MAX	Maximum ratio of effectively used space of the persistent volume size in percent

Column	Meaning
OBJECT_COUNT_AVG	Average number of schema objects in the database.
OBJECT_COUNT_MAX	Maximum number of schema objects in the database.

EXA_MONITOR_LAST_DAY

This system table describes monitoring information (the maximal values in the cluster).

 The data ratios are no indicators to the hardware performance. They were introduced to improve the comparability in case of variations of the measure intervals. If you multiply the ratio with the last interval duration, you get the real data volumes.

All users have access.

Column	Meaning
MEASURE_TIME	Point of the measurement
LOAD	System load (equals the load value of program uptime)
CPU	CPU utilization in percent (of the database instance, averaged over the last measure interval)
TEMP_DB_RAM	Usage of temporary DB memory in MiB (of the database instance, maximum over the last measure interval)
HDD_READ	Hard disk read ratio in MiB per second (per node, averaged over the last measure interval)
HDD_WRITE	Hard disk write ratio in MiB per second (per node, averaged over the last measure interval)
NET	Network traffic ratio in MiB per second (sum of send/receive, per node, averaged over the last measure interval)
SWAP	Swap ratio in MiB per second (averaged over the last measure interval). If this value is higher than 0, a system configuration problem may exist.

EXA_MONITOR_HOURLY

This system table describes the hourly aggregated monitoring information (of values from [EXA_MONITOR_LAST_DAY](#)) sorted by the interval start.

 The data ratios are no indicators to the hardware performance. They were introduced to improve the comparability in case of variations of the measure intervals. If you multiply the ratio with the last interval duration, you get the real data volumes.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
LOAD_AVG	Average system load (equals the 1-minute load value of program uptime)
LOAD_MAX	Maximal system load (equals the 1-minute load value of program uptime)
CPU_AVG	Average CPU utilization in percent (of the database instance)
CPU_MAX	Maximal CPU utilization in percent (of the database instance)
TEMP_DB_RAM_AVG	Average usage of temporary DB memory in MiB (of the database instance)
TEMP_DB_RAM_MAX	Maximal usage of temporary DB memory in MiB (of the database instance)
HDD_READ_AVG	Average hard disk read ratio in MiB per second

Column	Meaning
HDD_READ_MAX	Maximal hard disk read ratio in MiB per second
HDD_WRITE_AVG	Average hard disk write ratio in MiB per second
HDD_WRITE_MAX	Maximal hard disk write ratio in MiB per second
NET_AVG	Average network traffic ratio in MiB per second
NET_MAX	Maximal network traffic ratio in MiB per second
SWAP_AVG	Average swap ratio in MiB per second. If this value is higher than 0, a system configuration problem may exist.
SWAP_MAX	Maximal swap ratio in MiB per second. If this value is higher than 0, a system configuration problem may exist.

EXA_MONITOR_DAILY

This system table describes the daily aggregated monitoring information (of values from [EXA_MONITOR_LAST_DAY](#)) sorted by the interval start.



The data ratios are no indicators to the hardware performance. They were introduced to improve the comparability in case of variations of the measure intervals. If you multiply the ratio with the last interval duration, you get the real data volumes.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
LOAD_AVG	Average system load (equals the 1-minute load value of program uptime)
LOAD_MAX	Maximal system load (equals the 1-minute load value of program uptime)
CPU_AVG	Average CPU utilization in percent (of the database instance)
CPU_MAX	Maximal CPU utilization in percent (of the database instance)
TEMP_DB_RAM_AVG	Average usage of temporary DB memory in MiB (of the database instance)
TEMP_DB_RAM_MAX	Maximal usage of temporary DB memory in MiB (of the database instance)
HDD_READ_AVG	Average hard disk read ratio in MiB per second
HDD_READ_MAX	Maximal hard disk read ratio in MiB per second
HDD_WRITE_AVG	Average hard disk write ratio in MiB per second
HDD_WRITE_MAX	Maximal hard disk write ratio in MiB per second
NET_AVG	Average network traffic ratio in MiB per second
NET_MAX	Maximal network traffic ratio in MiB per second
SWAP_AVG	Average swap ratio in MiB per second. If this value is higher than 0, a system configuration problem may exist.
SWAP_MAX	Maximal swap ratio in MiB per second. If this value is higher than 0, a system configuration problem may exist.

EXA_MONITOR_MONTHLY

This system table describes the monthly aggregated monitoring information (of values from [EXA_MONITOR_LAST_DAY](#)) sorted by the interval start.

 The data ratios are no indicators to the hardware performance. They were introduced to improve the comparability in case of variations of the measure intervals. If you multiply the ratio with the last interval duration, you get the real data volumes.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
LOAD_AVG	Average system load (equals the 1-minute load value of program uptime)
LOAD_MAX	Maximal system load (equals the 1-minute load value of program uptime)
CPU_AVG	Average CPU utilization in percent (of the database instance)
CPU_MAX	Maximal CPU utilization in percent (of the database instance)
TEMP_DB_RAM_AVG	Average usage of temporary DB memory in MiB (of the database instance)
TEMP_DB_RAM_MAX	Maximal usage of temporary DB memory in MiB (of the database instance)
HDD_READ_AVG	Average hard disk read ratio in MiB per second
HDD_READ_MAX	Maximal hard disk read ratio in MiB per second
HDD_WRITE_AVG	Average hard disk write ratio in MiB per second
HDD_WRITE_MAX	Maximal hard disk write ratio in MiB per second
NET_AVG	Average network traffic ratio in MiB per second
NET_MAX	Maximal network traffic ratio in MiB per second
SWAP_AVG	Average swap ratio in MiB per second. If this value is higher than 0, a system configuration problem may exist.
SWAP_MAX	Maximal swap ratio in MiB per second. If this value is higher than 0, a system configuration problem may exist.

EXA_SQL_LAST_DAY

This system table contains all executed SQL statements without any reference to the executing user or detail sql texts. Only those statements are considered which could be successfully compiled.

All users have access.

Column	Meaning
SESSION_ID	Id of the session
STMT_ID	Serially numbered id of statement within a session
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)
DURATION	Duration of the statement in seconds
START_TIME	Start point of the statement
STOP_TIME	Stop point of the statement
CPU	CPU utilization in percent
TEMP_DB_RAM_PEAK	Maximal usage of temporary DB memory of the query in MiB (cluster wide)
HDD_READ	Maximal hard disk read ratio in MiB per second (per node, averaged over the last measure interval)
	 If this value is larger than 0, then data had to be loaded into the main memory.

Column	Meaning
HDD_WRITE	Maximal hard disk write ratio in MiB per second (per node, averaged over the last measure interval)  This column reflects only the data written during a COMMIT. For other statements its value is NULL.
NET	Maximal network traffic ratio in MiB per second (sum of send/receive, per node, averaged over the last measure interval)
SUCCESS	Result of the statement TRUE Statement was executed successfully FALSE Statement failed (e.g. with a data exception)
ERROR_CODE	Error code if the statement failed
ERROR_TEXT	Error text if the statement failed
PRIORITY	Priority group
NICE	NICE attribute
RESOURCES	Allocated resources in percent
ROW_COUNT	Number of result rows for queries, or number of affected rows for DML and DDL statements
EXECUTION_MODE	EXECUTE Normal execution of statements PREPARE Prepared phase for prepared statements CACHED Query which accesses the Query Cache PREPROCESS Execution of the Preprocessor script

EXA_SQL_HOURLY

This system table contains the hourly aggregated number of executed SQL statements sorted by the interval start. Per interval several entries for each command type (e.g. SELECT) is created.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)
SUCCESS	Result of the statement TRUE Statement was executed successfully FALSE Statement failed (e.g. with a data exception)
COUNT	Number of executions
DURATION_AVG	Average execution time of statements
DURATION_MAX	Maximal execution time of statements
CPU_AVG	Average CPU utilization in percent

Column	Meaning	
CPU_MAX	Maximal CPU utilization in percent	
TEMP_DB_RAM_PEAK_AVG	Average usage of temporary DB memory of queries in MiB (cluster wide)	
TEMP_DB_RAM_PEAK_MAX	Maximal usage of temporary DB memory of queries in MiB (cluster wide)	
HDD_READ_AVG	Average hard disk read ratio in MiB per second (per node)	
HDD_READ_MAX	Maximal hard disk read ratio in MiB per second (per node)	
HDD_WRITE_AVG	Average hard disk write ratio in MiB per second (per node, COMMIT only)	
HDD_WRITE_MAX	Maximal hard disk write ratio in MiB per second (per node, COMMIT only)	
NET_AVG	Average network traffic ratio in MiB per second (per node)	
NET_MAX	Maximal network traffic ratio in MiB per second (per node)	
ROW_COUNT_AVG	Average number of result rows for queries, or number of affected rows for DML and DDL statements	
ROW_COUNT_MAX	Maximal number of result rows for queries, or number of affected rows for DML and DDL statements	
EXECUTION_MODE	EXECUTE Normal execution of statements PREPARE Prepared phase for prepared statements CACHED Query which accesses the Query Cache PREPROCESS Execution of the Preprocessor script	

EXA_SQL_DAILY

This system table contains the daily aggregated number of executed SQL statements sorted by the interval start. Per interval several entries for each command type (e.g. SELECT) is created.

All users have access.

Column	Meaning	
INTERVAL_START	Start point of the aggregation interval	
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)	
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)	
SUCCESS	Result of the statement TRUE Statement was executed successfully FALSE Statement failed (e.g. with a data exception)	
COUNT	Number of executions	
DURATION_AVG	Average execution time of statements	
DURATION_MAX	Maximal execution time of statements	
CPU_AVG	Average CPU utilization in percent	
CPU_MAX	Maximal CPU utilization in percent	
TEMP_DB_RAM_PEAK_AVG	Average usage of temporary DB memory of queries in MiB (cluster wide)	
TEMP_DB_RAM_PEAK_MAX	Maximal usage of temporary DB memory of queries in MiB (cluster wide)	

Column	Meaning	
HDD_READ_AVG	Average hard disk read ratio in MiB per second (per node)	
HDD_READ_MAX	Maximal hard disk read ratio in MiB per second (per node)	
HDD_WRITE_AVG	Average hard disk write ratio in MiB per second (per node, COMMIT only)	
HDD_WRITE_MAX	Maximal hard disk write ratio in MiB per second (per node, COMMIT only)	
NET_AVG	Average network traffic ratio in MiB per second (per node)	
NET_MAX	Maximal network traffic ratio in MiB per second (per node)	
ROW_COUNT_AVG	Average number of result rows for queries, or number of affected rows for DML and DDL statements	
ROW_COUNT_MAX	Maximal number of result rows for queries, or number of affected rows for DML and DDL statements	
EXECUTION_MODE	EXECUTE Normal execution of statements PREPARE Prepared phase for prepared statements CACHED Query which accesses the Query Cache PREPROCESS Execution of the Preprocessor script	

EXA_SQL_MONTHLY

This system table contains the monthly aggregated number of executed SQL statements sorted by the interval start. Per interval several entries for each command type (e.g. SELECT) is created.

All users have access.

Column	Meaning	
INTERVAL_START	Start point of the aggregation interval	
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)	
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)	
SUCCESS	Result of the statement TRUE Statement was executed successfully FALSE Statement failed (e.g. with a data exception)	
COUNT	Number of executions	
DURATION_AVG	Average execution time of statements	
DURATION_MAX	Maximal execution time of statements	
CPU_AVG	Average CPU utilization in percent	
CPU_MAX	Maximal CPU utilization in percent	
TEMP_DB_RAM_PEAK_AVG	Average usage of temporary DB memory of queries in MiB (cluster wide)	
TEMP_DB_RAM_PEAK_MAX	Maximal usage of temporary DB memory of queries in MiB (cluster wide)	
HDD_READ_AVG	Average hard disk read ratio in MiB per second (per node)	
HDD_READ_MAX	Maximal hard disk read ratio in MiB per second (per node)	
HDD_WRITE_AVG	Average hard disk write ratio in MiB per second (per node, COMMIT only)	

Column	Meaning	
HDD_WRITE_MAX	Maximal hard disk write ratio in MiB per second (per node, COMMIT only)	
NET_AVG	Average network traffic ratio in MiB per second (per node)	
NET_MAX	Maximal network traffic ratio in MiB per second (per node)	
ROW_COUNT_AVG	Average number of result rows for queries, or number of affected rows for DML and DDL statements	
ROW_COUNT_MAX	Maximal number of result rows for queries, or number of affected rows for DML and DDL statements	
EXECUTION_MODE	EXECUTE Normal execution of statements PREPARE Prepared phase for prepared statements CACHED Query which accesses the Query Cache PREPROCESS Execution of the Preprocessor script	

EXA_SYSTEM_EVENTS

This system table contains system events like startup or shutdown of the DBMS.

All users have access.

Column	Meaning	
MEASURE_TIME	Time of the event	
EVENT_TYPE	STARTUP DBMS was started and logins are now possible SHUTDOWN DBMS was stopped BACKUP_START Start of a backup job BACKUP_END End of a backup job RESTORE_START Start of a restore job RESTORE_END End of a restore job FAILSAFETY Node failure with possible automatic DBMS restart using a stand by node RECOVERY_START Start of the data restore process RECOVERY_END End of the data restore process (at this point the full redundancy level of the cluster is reestablished) RESTART Restart of the DBMS due to a failure LICENSE_EXCEEDED License limit exceeded (databases reject inserting statements) LICENSE_OK License limit O.K. (databases allow inserting statements) SIZE_LIMIT_EXCEEDED Size limit exceeded (database rejects inserting statements) SIZE_LIMIT_OK Size limit O.K. (database allows inserting statements)	
DBMS_VERSION	Version of DBMS	
NODES	Number of cluster nodes	
DB_RAM_SIZE	Used DB RAM license in GiB	
PARAMETERS	Parameters for the DBMS	

EXA_USAGE_LAST_DAY

This system table contains information about the DBMS usage of the recent 24 hours.

All users have access.

Column	Meaning
MEASURE_TIME	Point of the measurement
USERS	Number of users connected to the DBMS
QUERIES	Number of concurrent queries

EXA_USAGE_HOURLY

This system table describes the hourly aggregated usage information of the DBMS, sorted by the interval start.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
USERS_AVG	Average number of users connected to the DBMS
USERS_MAX	Maximum number of users connected to the DBMS
QUERIES_AVG	Average number of concurrent queries
QUERIES_MAX	Maximum number of concurrent queries
IDLE	Percentage of last period where no query is running at all

EXA_USAGE_DAILY

This system table describes the daily aggregated usage information of the DBMS, sorted by the interval start.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
USERS_AVG	Average number of users connected to the DBMS
USERS_MAX	Maximum number of users connected to the DBMS
QUERIES_AVG	Average number of concurrent queries
QUERIES_MAX	Maximum number of concurrent queries
IDLE	Percentage of last period where no query is running at all

EXA_USAGE_MONTHLY

This system table describes the monthly aggregated usage information of the DBMS, sorted by the interval start.

All users have access.

Column	Meaning
INTERVAL_START	Start point of the aggregation interval
USERS_AVG	Average number of users connected to the DBMS
USERS_MAX	Maximum number of users connected to the DBMS

Column	Meaning
QUERIES_AVG	Average number of concurrent queries
QUERIES_MAX	Maximum number of concurrent queries
IDLE	Percentage of last period where no query is running at all

EXA_USER_PROFILE_LAST_DAY

Lists all profiling information of *own* sessions with activated profiling. Details for this topic can also be found in [Section 3.9, “Profiling”](#).

All users have access.

Column	Meaning										
SESSION_ID	Id of the session										
STMT_ID	Serially numbered id of statement within a session										
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)										
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)										
PART_ID	Serially numbered id of the execution part within the statement										
PART_NAME	Name of the execution part (see also Section 3.9, “Profiling”)										
PART_INFO	Extended information of the execution part <table> <tr> <td>GLOBAL</td><td>Global action (e.g. global join)</td></tr> <tr> <td>EXPRESSION INDEX</td><td>Non-persistent join index which is built on an expression</td></tr> <tr> <td>NL JOIN</td><td>Nested loop join (cross product)</td></tr> <tr> <td>REPLICATED</td><td>Replicated object (e.g. small tables)</td></tr> <tr> <td>TEMPORARY</td><td>Temporary object (for intermediate results)</td></tr> </table>	GLOBAL	Global action (e.g. global join)	EXPRESSION INDEX	Non-persistent join index which is built on an expression	NL JOIN	Nested loop join (cross product)	REPLICATED	Replicated object (e.g. small tables)	TEMPORARY	Temporary object (for intermediate results)
GLOBAL	Global action (e.g. global join)										
EXPRESSION INDEX	Non-persistent join index which is built on an expression										
NL JOIN	Nested loop join (cross product)										
REPLICATED	Replicated object (e.g. small tables)										
TEMPORARY	Temporary object (for intermediate results)										
OBJECT_SCHEMA	Schema of the processed object										
OBJECT_NAME	Name of the processed object										
OBJECT_ROWS	Number of rows of the processed object										
OUT_ROWS	Number of result rows of the execution part										
DURATION	Duration of the execution part in seconds										
CPU	CPU utilization in percent of the execution part (averaged over the duration)										
TEMP_DB_RAM_PEAK	Usage of temporary DB memory of the execution part in MiB (cluster wide, maximum over the duration)										
HDD_READ	Hard disk read ratio in MiB per second (per node, averaged over the duration) <p> If this value is larger than 0, then data had to be loaded into the main memory.</p>										
HDD_WRITE	Hard disk write ratio in MiB per second (per node, averaged over the duration) <p> This column reflects only the data written during a COMMIT. For other statements its value is NULL.</p>										
NET	Network traffic ratio in MiB per second (sum of send/receive, per node, averaged over the duration)										
REMARKS	Additional information										
SQL_TEXT	Corresponding SQL text										

EXA_USER_PROFILE_RUNNING

Lists all profiling information of your *own* running queries. Details for this topic can also be found in [Section 3.9, “Profiling”](#).

All users have access.

Column	Meaning										
SESSION_ID	Id of the session										
STMT_ID	Serially numbered id of statement within a session										
COMMAND_NAME	Name of the statement (e.g. SELECT, COMMIT, MERGE etc.)										
COMMAND_CLASS	Class of statement (e.g. DQL, TRANSACTION, DML etc.)										
PART_ID	Serially numbered id of the execution part within the statement										
PART_NAME	Name of the execution part (see also Section 3.9, “Profiling”)										
PART_INFO	Extended information of the execution part <table> <tr> <td>GLOBAL</td><td>Global action (e.g. global join)</td></tr> <tr> <td>EXPRESSION INDEX</td><td>Non-persistent join index which is built on an expression</td></tr> <tr> <td>NL JOIN</td><td>Nested loop join (cross product)</td></tr> <tr> <td>REPLICATED</td><td>Replicated object (e.g. small tables)</td></tr> <tr> <td>TEMPORARY</td><td>Temporary object (for intermediate results)</td></tr> </table>	GLOBAL	Global action (e.g. global join)	EXPRESSION INDEX	Non-persistent join index which is built on an expression	NL JOIN	Nested loop join (cross product)	REPLICATED	Replicated object (e.g. small tables)	TEMPORARY	Temporary object (for intermediate results)
GLOBAL	Global action (e.g. global join)										
EXPRESSION INDEX	Non-persistent join index which is built on an expression										
NL JOIN	Nested loop join (cross product)										
REPLICATED	Replicated object (e.g. small tables)										
TEMPORARY	Temporary object (for intermediate results)										
PART_FINISHED	Defines whether the execution part has already been finished										
OBJECT_SCHEMA	Schema of the processed object										
OBJECT_NAME	Name of the processed object										
OBJECT_ROWS	Number of rows of the processed object										
OUT_ROWS	Number of result rows of the execution part										
DURATION	Duration of the execution part in seconds										
CPU	CPU utilization in percent of the execution part (averaged over the duration)										
TEMP_DB_RAM_PEAK	Usage of temporary DB memory of the execution part in MiB (cluster wide, maximum over the duration)										
HDD_READ	Hard disk read ratio in MiB per second (per node, averaged over the duration) <p> If this value is larger than 0, then data had to be loaded into the main memory.</p>										
HDD_WRITE	Hard disk write ratio in MiB per second (per node, averaged over the duration) <p> This column reflects only the data written during a COMMIT. For other statements its value is NULL.</p>										
NET	Network traffic ratio in MiB per second (sum of send/receive, per node, averaged over the duration)										
REMARKS	Additional information										
SQL_TEXT	Corresponding SQL text										

EXA_USER_SESSIONS_LAST_DAY

Lists all *own* sessions of the last day.

All users have access.

Column	Meaning
SESSION_ID	Id of the session
LOGIN_TIME	Time of login
LOGOUT_TIME	Time of logout
USER_NAME	User name
CLIENT	Client application used by the user
DRIVER	Used driver
ENCRYPTED	Flag whether the connection is encrypted
HOST	Computer name or IP address from which the user has logged-in
OS_USER	User name under which the user logged into the operating system of the computer from which the login came
OS_NAME	Operating system of the client server
SUCCESS	Information whether the login was successful
ERROR_CODE	Error code if the login failed
ERROR_TEXT	Error text if the login failed

EXA_USER_TRANSACTION_CONFLICTS_LAST_DAY

Lists all transaction conflicts linked to the current user's sessions.

This system table can be cleared by the statement [TRUNCATE AUDIT LOGS](#).

All users have access.

Column	Meaning
SESSION_ID	Id of the session
CONFLICT_SESSION_ID	Session which produces the conflict
START_TIME	Start time of the conflict
STOP_TIME	End time of the conflict or NULL if the conflict is still open
CONFLICT_TYPE	Type of the conflict: WAIT FOR COMMIT One session has to wait until the other one is committed TRANSACTION ROLL-BACK One session has to be rolled back due to a conflict
CONFLICT_OBJECTS	Name of the corresponding objects
CONFLICT_INFO	Additional information about the conflict

A.2.4. System tables compatible with Oracle

The following system tables were implemented in order to provide Oracle-related functionality.

CAT

This system table lists all of the tables and views in the current schema. CAT is not completely compatible with Oracle because here all objects owned by the user are visible. This definition was decided upon because in Exasol several schemas can belong to one user.

All users have access.

Column	Meaning
TABLE_NAME	Name of the table or the view
TABLE_TYPE	Type of object: TABLE or VIEW

DUAL

Along the lines of Oracle's identically named system table, this system table can be used to output static information (e.g. "SELECT CURRENT_USER FROM DUAL"). It contains one row with one single column.

All users have access.

Column	Meaning
DUMMY	Contains a NULL value

Appendix B. Details on rights management

In this chapter all of the details important to rights management are explained. This includes a list of all system and object privileges as well as a list of the system tables available for these.

An introduction to the basic concepts of rights management can be found in [Section 3.2, “Rights management”](#). Further details on the various SQL statements can be found in section [Section 2.2.3, “Access control using SQL \(DCL\)”](#).

B.1. List of system and object privileges

The following two tables contain all of the defined system and object privileges in Exasol.

Table B.1. System privileges in Exasol

System privilege	Permitted actions
MISC	
GRANT ANY OBJECT PRIVILEGE	Grant or withdraw any object rights
GRANT ANY PRIVILEGE	Grant or withdraw any system rights (this is a powerful privilege and should only be granted to a small number of users)
GRANT ANY PRIORITY	Grant or withdraw a priority
CREATE SESSION	Connect to database
KILL ANY SESSION	Kill session or query
ALTER SYSTEM	Alter system-wide settings (e.g. NLS_DATE_FORMAT)
USERS	
CREATE USER	Create user
ALTER USER	Alter the password of any user (this is a powerful privilege and should only be granted to a small number of users)
DROP USER	Delete user
ROLES	
CREATE ROLE	Create roles
DROP ANY ROLE	Delete roles
GRANT ANY ROLE	Grant any role (this is a powerful privilege and should only be granted to a small number of users)
CONNECTIONS	
CREATE CONNECTION	Create external connections
ALTER ANY CONNECTION	Change connection data of a connection
DROP ANY CONNECTION	Delete connections
GRANT ANY CONNECTION	Grant any connection to users/roles
USE ANY CONNECTION	Using any connection in statements IMPORT and EXPORT
ACCESS ANY CONNECTION	Access any connection details from scripts
SCHEMAS	
CREATE SCHEMA	Creation of a schema
ALTER ANY SCHEMA	Allocate a schema to another user or role
DROP ANY SCHEMA	Delete any schema
CREATE VIRTUAL SCHEMA	Creation of a virtual schema
ALTER ANY VIRTUAL SCHEMA	Update parameters of any virtual schema
ALTER ANY VIRTUAL SCHEMA REFRESH	Update the metadata of any virtual schema
DROP ANY VIRTUAL SCHEMA	Delete any virtual schema
TABLES	
CREATE TABLE	Create table in one's own schema or that of an allocated role (DROP is implicit as the owner of a table can always delete these)
CREATE ANY TABLE	Create a table in any schema
ALTER ANY TABLE	Alter a table in any schema
DELETE ANY TABLE	Delete rows in a table in any schema
DROP ANY TABLE	Delete a table in any schema
INSERT ANY TABLE	Insert data into a table from any schema

System privilege	Permitted actions
SELECT ANY TABLE	Access the contents of a table or view from any schema (does not include system tables)
SELECT ANY DICTIONARY	Access the contents of any system table
UPDATE ANY TABLE	Alter rows in a table from any schema
VIEWS	
CREATE VIEW	Create views in one's own schema or in that of an allocated role (DROP is implicit as the owner of a view can always delete these)
CREATE ANY VIEW	Create a view in any schema
DROP ANY VIEW	Delete a view in any schema
FUNCTIONS	
CREATE FUNCTION	Create functions in one's own schema or in those of an allocated role (DROP is implicit as the owner of a function can always delete these)
CREATE ANY FUNCTION	Create functions in any schema
DROP ANY FUNCTION	Delete functions from any schema
EXECUTE ANY FUNCTION	Execute functions from any schema
SCRIPTS	
CREATE SCRIPT	Create scripts in one's own schema or in those of an allocated role (DROP is implicit as the owner of a script can always delete these)
CREATE ANY SCRIPT	Create scripts in any schema
DROP ANY SCRIPT	Delete scripts from any schema
EXECUTE ANY SCRIPT	Execute scripts from any schema

Table B.2. Object privileges in Exasol

Object privilege	Schema objects	Permitted actions
ALTER	Schema, table, virtual schema	Run the ALTER TABLE statement
SELECT	Schema, table, view, virtual schema, virtual table	Access to the table contents
INSERT	Schema, table	Insert rows in a table
UPDATE	Schema, table	Change the contents of a row in a table
DELETE	Schema, table	Deletion of rows
REFERENCES	Table	Creation of foreign keys referencing this table
EXECUTE	Schema, function, script	Run functions or scripts
ACCESS	Connection	Access details of a connection from scripts
REFRESH	Virtual schema	Update the metadata of a virtual schema

B.2. Required privileges for SQL statements

The following table lists all of the SQL statements supported by Exasol as well as the necessary privileges.

Table B.3. Required privileges for running SQL statements

SQL statement	Required privileges
USERS	
CREATE USER my_user ...	System privilege CREATE USER
ALTER USER my_user ...	One's own password can always be changed. The ALTER USER system privilege is needed for other users.
DROP USER my_user;	System privilege DROP USER
ROLES	
CREATE ROLE r ...	System privilege CREATE ROLE
DROP ROLE r;	System privilege DROP ROLE or one has to receive this role with the ADMIN OPTION.
CONNECTIONS	
CREATE CONNECTION my_connection ...	System privilege CREATE CONNECTION
ALTER CONNECTION my_connection ...	System privilege ALTER ANY CONNECTION or the connection has to be granted to you with the ADMIN OPTION.
DROP CONNECTION my_connection;	System privilege DROP ANY CONNECTION or the connection has to be granted to you with the ADMIN OPTION.
IMPORT ... FROM my_connection ...	System privilege USE ANY CONNECTION or the connection has to be granted to you.
SCHEMAS	
CREATE SCHEMA myschema;	System privilege CREATE SCHEMA
OPEN SCHEMA myschema;	No restriction. Note: the schema objects within the schema are not automatically readable!
DROP SCHEMA myschema;	System privilege DROP ANY SCHEMA or the schema is owned by the current user. If CASCADE is specified, all of the schema objects contained in the schema will also be deleted!
ALTER SCHEMA myschema CHANGE OWNER userOrRole;	System privilege ALTER ANY SCHEMA
CREATE VIRTUAL SCHEMA s	System privilege CREATE VIRTUAL SCHEMA
DROP VIRTUAL SCHEMA s	System privilege DROP ANY VIRTUAL SCHEMA or s is owned by the current user
ALTER VIRTUAL SCHEMA s SET ...	System privilege ALTER ANY VIRTUAL SCHEMA, object privilege ALTER on s or the schema is owned by the current user. Additionally, access rights are necessary on the corresponding adapter script.
ALTER VIRTUAL SCHEMA s REFRESH	System privileges ALTER ANY VIRTUAL SCHEMA or ALTER ANY VIRTUAL SCHEMA REFRESH, object privilege ALTER or REFRESH on s, or s is owned by the current user. Additionally, access rights are necessary on the corresponding adapter script.
ALTER VIRTUAL SCHEMA s CHANGE OWNER u	System privilege ALTER ANY VIRTUAL SCHEMA or s is owned by the current user. Note: object privilege ALTER is not sufficient.
OPEN SCHEMA s	No restriction.
TABLES	

SQL statement	Required privileges
CREATE TABLE t (<col_defs>)	System privilege CREATE TABLE if the table is in one's own schema or that of an assigned role. Otherwise, system privilege CREATE ANY TABLE.
CREATE TABLE AS <subquery>	Similar to CREATE TABLE t (<col_defs>) but the user must also possess SELECT privileges on the tables of the subquery.
CREATE OR REPLACE TABLE t ...	Similar to CREATE TABLE t (<col_defs>) but if the table is replaced, the user must also possess the necessary privileges, such as for DROP TABLE t.
ALTER TABLE t ...	System privilege ALTER ANY TABLE, object privilege ALTER on t or its schema or t is owned by the current user or one of that user's roles.
SELECT * FROM t;	System privilege SELECT ANY TABLE or object privilege SELECT on t or its schema or t is owned by the current user or one of that user's roles. If the table is part of a virtual schema, appropriate access rights are necessary on the corresponding adapter script and the indirectly used connections.
INSERT INTO t ...	System privilege INSERT ANY TABLE or object privilege INSERT on t or its schema or t is owned by the current user or one of that user's roles.
UPDATE t SET ...;	System privilege UPDATE ANY TABLE or object privilege UPDATE on t or its schema or t is owned by the current user or one of that user's roles.
MERGE INTO t USING u ...	Corresponding INSERT and UPDATE privileges on t as well as SELECT privileges on u.
DELETE FROM t;	System privilege DELETE ANY TABLE or object privilege DELETE on t or t is owned by the current user or one of that user's roles.
TRUNCATE TABLE t;	System privilege DELETE ANY TABLE or object privilege DELETE on t or t is owned by the current user or one of that user's roles.
DROP TABLE t;	System privilege DROP ANY TABLE or t is owned by the current user or one of that user's roles.
RECOMPRESS TABLE t;	Access to the table by any of the modifying ANY TABLE system privileges, any modifying object privilege (that means any except SELECT), or the object is owned by the user or any of its roles.
REORGANIZE TABLE t;	Access to the table by any of the modifying ANY TABLE system privileges, any modifying object privilege (that means any except SELECT), or the object is owned by the user or any of its roles.
PRELOAD TABLE t;	Access to the table by any read/write system or object privilege, or the object is owned by the user or any of its roles.
<i>Create foreign key on t</i>	Object privilege REFERENCES on t or t is owned by the current user or one of that user's roles.
VIEWS	

SQL statement	Required privileges
CREATE VIEW v AS ...	System privilege CREATE VIEW if the view is in one's own schema or that of an assigned role. Otherwise, system privilege CREATE ANY VIEW. Additionally, the owner of the view (who is not automatically the CREATOR) must possess the corresponding SELECT privileges on all the referenced base tables.
CREATE OR REPLACE VIEW v ...	Similar to CREATE VIEW but if the view is replaced, the user must also possess the necessary privileges, such as for DROP VIEW v.
SELECT * FROM v;	System privilege SELECT ANY TABLE or object privilege SELECT on v or its schema. Additionally, the owner of v must possess the corresponding SELECT privileges on the referenced base tables of the view. If the view contains a table from a virtual schema, access rights are necessary on the corresponding adapter script. If the view accesses objects of a virtual schema, appropriate access rights are necessary on the corresponding adapter script and the indirectly used connections by the user invoking the SELECT statement.
DROP VIEW v;	System privilege DROP ANY VIEW or v is owned by the current user or one of that user's roles.
FUNCTIONS	
CREATE FUNCTION f ...	System privilege CREATE FUNCTION if the function is in one's own schema or that of an assigned role. Otherwise, system privilege CREATE ANY FUNCTION.
CREATE OR REPLACE FUNCTION f ...	Similar to CREATE FUNCTION but if the function is replaced, the user must also possess the necessary privileges, such as for DROP FUNCTION f.
SELECT f(...) FROM t;	System privilege EXECUTE ANY FUNCTION or object privilege EXECUTE on the function or its schema.
DROP FUNCTION f;	System privilege DROP ANY FUNCTION if function f is not in one's own schema or that of an assigned role.
SCRIPTS	
CREATE SCRIPT s ...	System privilege CREATE SCRIPT if the script is in one's own schema or that of an assigned role. Otherwise, system privilege CREATE ANY SCRIPT.
CREATE OR REPLACE SCRIPT s ...	Similar to CREATE SCRIPT but if the script is replaced, the user must also possess the necessary privileges, such as for DROP SCRIPT s.
EXECUTE SCRIPT s;	System privilege EXECUTE ANY SCRIPT or object privilege EXECUTE on the script or its schema.
DROP SCRIPT s;	System privilege DROP ANY SCRIPT if script s is not in one's own schema or that of an assigned role.
RENAME	
RENAME o TO x;	If o is a schema, the schema must belong to the user or one of that user's roles. If o is a schema object, the object must belong to the user or one of that user's roles (i.e. located in one's own schema or that of an assigned role). If o is a user, role or connection, then the user must have the corresponding system privileges.
GRANT	

SQL statement	Required privileges
GRANT <sys_priv> TO u	System privilege GRANT ANY PRIVILEGE or the user must have received this system privilege with the WITH ADMIN OPTION.
GRANT <ob_priv> ON o TO u	System privilege GRANT ANY OBJECT PRIVILEGE or the user or one of that user's roles must own schema object o.
GRANT r TO u	System privilege GRANT ANY ROLE or the user must have received this role with the WITH ADMIN OPTION.
GRANT PRIORITY c TO u	System privilege GRANT ANY PRIORITY.
GRANT CONNECTION c TO u	System privilege GRANT ANY CONNECTION or the user must have received this connection with the WITH ADMIN OPTION
REVOKE	
REVOKE <sys_priv> FROM u	System privilege GRANT ANY PRIVILEGE or the user must have received this system privilege with the WITH ADMIN OPTION.
REVOKE <ob_priv> ON o FROM u	System privilege GRANT ANY OBJECT PRIVILEGE or the user must self-grant this object privilege.
REVOKE r FROM u	System privilege GRANT ANY ROLE or the user must have received this role with the WITH ADMIN OPTION.
REVOKE PRIORITY FROM u	System privilege GRANT ANY PRIORITY.
REVOKE CONNECTION c TO u	System privilege GRANT ANY CONNECTION or the user must have received this connection with the WITH ADMIN OPTION
MISC	
ALTER SESSION ...	No privileges are needed for this statement.
KILL SESSION ...	System privilege KILL ANY SESSION if it's not your own session.
ALTER SYSTEM ...	System privilege ALTER SYSTEM.
DESCRIBE o	Schema objects can be described with the DESCRIBE statement if the user or one of the user's roles is the owner of or has access to that object (object or system privileges).

B.3. System tables for rights management

The following system tables are available for information on the state of rights management in Exasol. A detailed description of all system tables can be found in [Appendix A, System tables](#).

User

EXA_DBAL_USERS	All users of the database
EXA_ALL_USERS	All users of the database, restricted information
EXA_USER_USERS	Current user

Roles

EXA_DBAL_ROLES and EXA_ALL_ROLES	All roles of the database
EXA_DBAL_ROLE_PRIVS	Granted roles
EXA_USER_ROLE_PRIVS	Roles directly granted to the current user

EXA_ROLE_ROLE_PRIVS	Roles possessed by the current user indirectly via other roles
EXA_SESSION_ROLES	Roles possessed by the current user

Connections

EXA_DBA_CONNECTIONS	All connections of the database
EXA_ALL_CONNECTIONS	All connections of the database, restricted information
EXA_DBA_CONNECTION_PRIVS	Granted connections
EXA_USER_CONNECTION_PRIVS	Connections directly granted to the current user
EXA_ROLE_CONNECTION_PRIVS	Connections possessed by the current user indirectly via other roles
EXA_SESSION_CONNECTIONS	Connection to which the current user has access

System privileges

EXA_DBA_SYS_PRIVS	Granted system privileges
EXA_USER_SYS_PRIVS	System privileges directly granted to the current user
EXA_ROLE_SYS_PRIVS	System privileges granted to the roles of the current user
EXA_SESSION_PRIVS	System privileges currently available to the user

Object privileges

EXA_DBA_OBJ_PRIVS and EXA_DBA_RESTRICTED_OBJ_PRIVS	Object privileges granted to objects on the database
EXA_ALL_OBJ_PRIVS	Similar to EXA_DBA_OBJ_PRIVS , but only for accessible objects of the database
EXA_USER_OBJ_PRIVS and EXA_USER_RESTRICTED_OBJ_PRIVS	Object privileges on the objects to which the current user has access apart from via the PUBLIC role
EXA_ROLE_OBJ_PRIVS and EXA_ROLE_RESTRICTED_OBJ_PRIVS	Object privileges that have been granted to the roles of the user
EXA_ALL_OBJ_PRIVS_MADE	Object privileges self-granted by the current user or those concerning that user's objects
EXA_USER_OBJ_PRIVS_MADE	Object privileges that relate to objects of the current user
EXA_ALL_OBJ_PRIVS_REC'D	Object privileges that have been directly granted to the current user or via PUBLIC
EXA_USER_OBJ_PRIVS_REC'D	Object privileges that have been directly granted to the current user

Appendix C. Compliance to the SQL standard

This section lists the parts of the current [SQL](#) standard (ISO/IEC 9075:2008) supported by Exasol. The SQL standard distinguishes between mandatory and optional features.

C.1. SQL 2008 Standard Mandatory Features

In the following table all of the mandatory features are listed.

Although many manufacturers claim compliance to all of the standards in their advertising, we are not aware of any system that actually supports all of the mandatory features of the SQL standard.

Symbol meaning:

- ✓ Fully supported
- ✓ Partially supported (i.e. not all sub-features are supported)

Table C.1. SQL 2008 Mandatory Features

Feature	Exasol
E011 Numeric data types	✓
E011-01 INTEGER and SMALLINT data types	✓
E011-02 REAL, DOUBLE PRECISION and FLOAT data types	✓
E011-03 DECIMAL and NUMERIC data types	✓
E011-04 Arithmetic operators	✓
E011-05 Numeric comparison	✓
E011-06 Implicit casting among numeric data types	✓
E021 Character string types	✓
E021-01 CHARACTER data type	✓
E021-02 CHARACTER VARYING data type	✓
E021-03 Character literals	✓
E021-04 CHARACTER_LENGTH function	✓
E021-05 OCTET_LENGTH function	✓
E021-06 SUBSTRING function	✓
E021-07 Character concatenation	✓
E021-08 UPPER and LOWER function	✓
E021-09 TRIM function	✓
E021-10 Implicit casting among the fixed-length and variable-length character string types	✓
E021-11 POSITION function	✓
E021-12 Character comparison	✓
E031 Identifiers	✓
E031-01 Delimited identifiers	✓
E031-02 Lower case identifiers	✓
E031-03 Trailing underscore	✓
E051 Basic query specification	✓
E051-01 SELECT DISTINCT	✓
E051-02 GROUP BY clause	✓
E051-04 GROUP BY can contain columns not in <select list>	✓
E051-05 Select list items can be renamed	✓
E051-06 HAVING clause	✓
E051-07 Qualified * in select list	✓
E051-08 Correlation names in the FROM clause	✓
E051-09 Rename columns in the FROM clause	✓
E061 Basic predicates and search conditions	✓
E061-01 Comparison predicate	✓
E061-02 BETWEEN predicate	✓
E061-03 IN predicate with list of values	✓
E061-04 LIKE predicate	✓
E061-05 LIKE predicate: ESCAPE clause	✓
E061-06 NULL predicate	✓

Feature	Exasol
E061-07 Quantified comparison predicate	
E061-08 EXISTS predicate	✓
E061-09 Subqueries in comparison predicate	✓
E061-11 Subqueries in IN predicate	✓
E061-12 Subqueries in quantified comparison predicate	
E061-13 Correlated Subqueries	✓
E061-14 Search condition	✓
E071 Basic query expressions	✓
E071-01 UNION DISTINCT table operator	✓
E071-02 UNION ALL table operator	✓
E071-03 EXCEPT DISTINCT table operator	✓
E071-05 Columns combined via table operators need not have exactly the same data type.	✓
E071-06 Table operators in subqueries	✓
E081 Basic privileges	✓
E081-01 SELECT privilege at the table level	✓
E081-02 DELETE privilege	✓
E081-03 INSERT privilege at the table level	✓
E081-04 UPDATE privilege at the table level	✓
E081-05 UPDATE privilege at the column level	
E081-06 REFERENCES privilege at the table level	✓
E081-07 REFERENCES privilege at the column level	
E081-08 WITH GRANT OPTION	
E081-09 USAGE privilege	
E081-10 EXECUTE privilege	✓
E091 Set functions	✓
E091-01 AVG	✓
E091-02 COUNT	✓
E091-03 MAX	✓
E091-04 MIN	✓
E091-05 SUM	✓
E091-06 ALL quantifier	✓
E091-07 DISTINCT quantifier	✓
E101 Basic data manipulation	✓
E101-01 INSERT statement	✓
E101-03 Searched UPDATE statement	✓
E101-04 Searched DELETE statement	✓
E111 Single row SELECT statement	
E121 Basic cursor support	
E121-01 DECLARE CURSOR	
E121-02 ORDER BY columns need not be in select list	
E121-03 Value expressions in ORDER BY clause	
E121-04 OPEN statement	

Feature	Exasol
E121-06 Positioned UPDATE statement	
E121-07 Positioned DELETE statement	
E121-08 CLOSE statement	
E121-10 FETCH statement: implicit NEXT	
E121-17 WITH HOLD cursors	
E131 Null value support (nulls in lieu of values)	✓
E141 Basic integrity constraints	✓
E141-01 NOT NULL constraint	✓
E141-02 UNIQUE constraints of NOT NULL columns	
E141-03 PRIMARY KEY constraint	✓
E141-04 Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action.	✓
E141-06 CHECK constraint	
E141-07 Column defaults	✓
E141-08 NOT NULL inferred on PRIMARY KEY	✓
E141-10 Names in a foreign key can be specified in any order	
E151 Transaction support	✓
E151-01 COMMIT statement	✓
E151-02 ROLLBACK statement	✓
E152 Basic SET TRANSACTION statement	
E152-01 SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02 SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E153 Updatable queries with subqueries	
E161 SQL comments using leading double minus	✓
E171 SQLSTATE support	
E182 Module language	
F031 Basic schema manipulation	✓
F031-01 CREATE TABLE statement to create persistent base tables	✓
F031-02 CREATE VIEW statement	✓
F031-03 GRANT statement	✓
F031-04 ALTER TABLE statement: ADD COLUMN clause	✓
F031-13 DROP TABLE statement: RESTRICT clause	✓
F031-16 DROP VIEW statement: RESTRICT clause	✓
F031-19 REVOKE statement: RESTRICT clause	✓
F041 Basic joined table	✓
F041-01 Inner join (but not necessarily the INNER keyword)	✓
F041-02 INNER keyword	✓
F041-03 LEFT OUTER JOIN	✓
F041-04 RIGHT OUTER JOIN	✓
F041-05 Outer joins can be nested	✓
F041-07 The inner table in a left or right outer join can also be used in an inner join	✓
F041-08 All comparison operators are supported (rather than just =)	✓

Feature	Exasol
F051 Basic date and time	✓
F051-01 DATE data type (including support of DATE literal)	✓
F051-02 TIME data type (including the support of TIME literal) with fractional seconds precision of at least 0	
F051-03 TIMESTAMP data type (including the support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	✓
F051-04 Comparison predicate on DATE, TIME and TIMESTAMP data types	✓
F051-05 Explicit CAST between datetime types and character string types	✓
F051-06 CURRENT_DATE	✓
F051-07 LOCALTIME	
F051-08 LOCALTIMESTAMP	✓
F081 UNION and EXCEPT in views	✓
F131 Grouped operations	✓
F131-01 WHERE, GROUP BY and HAVING clauses supported in queries with grouped views	✓
F131-02 Multiple tables supported in queries with grouped views	✓
F131-03 Set functions supported in queries with grouped views	✓
F131-04 Subqueries with GROUP BY and HAVING clauses and grouped views	✓
F131-05 Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F181 Multiple module support	
F201 CAST function	✓
F221 Explicit defaults	✓
F261 CASE expression	✓
F261-01 Simple CASE	✓
F261-02 Searched CASE	✓
F261-03 NULLIF	✓
F261-04 COALESCE	✓
F311 Schema definition statement	✓
F311-01 CREATE SCHEMA	✓
F311-02 CREATE TABLE for persistent base tables (within CREATE SCHEMA)	
F311-03 CREATE VIEW (within CREATE SCHEMA)	
F311-04 CREATE VIEW: WITH CHECK OPTION (within CREATE SCHEMA)	
F311-05 GRANT STATEMENT (within CREATE SCHEMA)	
F471 Scalar subquery values	✓
F481 Expanded NULL predicate	✓
F812 Basic flagging	
S011 Distinct data types	
T321 Basic SQL-invoked routines	✓
T321-01 User-defined functions with no overloading	✓
T321-02 User-defined stored procedures with no overloading	
T321-03 Function invocation	✓
T321-04 CALL statement	
T321-05 RETURN statement	✓

Feature	Exasol
T631 IN predicate with one list element	✓

C.2. SQL 2008 Standard Optional Features

Table C.2. SQL 2008 Optional Features supported by Exasol

Feature ID	Feature
F033	ALTER TABLE statement: DROP COLUMN clause
F052	Intervals and datetime arithmetic
F171	Multiple schemas per user
F222	INSERT statement: DEFAULT VALUES clause
F302-01	INTERSECT DISTINCT table operator
F312	MERGE statement
F321	User authorization
F381	Extended schema manipulation
F381-01	ALTER TABLE statement: ALTER COLUMN clause
F381-02	ALTER TABLE statement: ADD CONSTRAINT clause
F381-03	ALTER TABLE statement: DROP CONSTRAINT clause
F391	Long identifiers
F401-02	Extended joined table: FULL OUTER JOIN
F401-04	Extended joined table: CROSS JOIN
F641	Row and table constructors
T031	BOOLEAN data type
T121	WITH (excluding RECURSIVE) in query expression
T171	LIKE clause in table definition
T172	AS subquery clause in table definition
T173	Extended LIKE clause in table definition
T174	Identity columns
T331	Basic roles
T351	Bracketed SQL comments /* ... */ comments)
T431	Extended grouping capabilities
T432	Nested and concatenated GROUPING SETS
T433	Multiargument GROUPING function
T434	GROUP BY DISTINCT
T441	ABS and MOD functions
T461	Symmetric BETWEEN predicate
T551	Optional key words for default syntax
T621	Enhanced numeric functions

Appendix D. Supported Encodings for ETL processes and EXAplus

In this section all supported encodings for ETL processes via [IMPORT](#) and [EXPORT](#) and EXAplus (see also [SET ENCODING](#) for usage) are listed.

Encoding	Aliases
ASCII	US-ASCII, US, ISO-IR-6, ANSI_X3.4-1968, ANSI_X3.4-1986, ISO_646.IRV:1991, ISO646-US, IBM367, IBM-367, CP367, CP-367, 367
ISO-8859-1	ISO8859-1, ISO88591, LATIN-1, LATIN1, L1, ISO-IR-100, ISO_8859-1:1987, ISO_8859-1, IBM819, IBM-819, CP819, CP-819, 819
ISO-8859-2	ISO8859-2, ISO88592, LATIN-2, LATIN2, L2, ISO-IR-101, ISO_8859-2:1987, ISO_8859-2
ISO-8859-3	ISO8859-3, ISO88593, LATIN-3, LATIN3, L3, ISO-IR-109, ISO_8859-3:1988, ISO_8859-3
ISO-8859-4	ISO8859-4, ISO88594, LATIN-4, LATIN4, L4, ISO-IR-110, ISO_8859-4:1988, ISO_8859-4
ISO-8859-5	ISO8859-5, ISO88595, CYRILLIC, ISO-IR-144, ISO_8859-5:1988, ISO_8859-5
ISO-8859-6	ISO8859-6, ISO88596, ARABIC, ISO-IR-127, ISO_8859-6:1987, ISO_8859-6, ECMA-114, ASMO-708
ISO-8859-7	ISO8859-7, ISO88597, GREEK, GREEK8, ISO-IR-126, ISO_8859-7:1987, ISO_8859-7, EL0T_928, ECMA-118
ISO-8859-8	ISO8859-8, ISO88598, HEBREW, ISO-IR-138, ISO_8859-8:1988, ISO_8859-8
ISO-8859-9	ISO8859-9, ISO88599, LATIN-5, LATIN5, L5, ISO-IR-148, ISO_8859-9:1989, ISO_8859-9
ISO-8859-11	ISO8859-11, ISO885911
ISO-8859-13	ISO8859-13, ISO885913, LATIN-7, LATIN7, L7, ISO-IR-179
ISO-8859-15	ISO8859-15, ISO885915, LATIN-9, LATIN9, L9
IBM850	IBM-850, CP850, CP-850, 850
IBM852	IBM-852, CP852, CP-852, 852
IBM855	IBM-855, CP855, CP-855, 855
IBM856	IBM-856, CP856, CP-856, 856
IBM857	IBM-857, CP857, CP-857, 857
IBM860	IBM-860, CP860, CP-860, 860
IBM861	IBM-861, CP861, CP-861, 861, CP-IS
IBM862	IBM-862, CP862, CP-862, 862
IBM863	IBM-863, CP863, CP-863, 863
IBM864	IBM-864, CP864, CP-864, 864
IBM865	IBM-865, CP865, CP-865, 865
IBM866	IBM-866, CP866, CP-866, 866
IBM868	IBM-868, CP868, CP-868, 868, CP-AR
IBM869	IBM-869, CP869, CP-869, 869, CP-GR
WINDOWS-1250	CP1250, CP-1250, 1250, MS-EE
WINDOWS-1251	CP1251, CP-1251, 1251, MS-CYRL

Encoding	Aliases
WINDOWS-1252	CP1252, CP-1252, 1252, MS-ANSI
WINDOWS-1253	CP1253, CP-1253, 1253, MS-GREEK
WINDOWS-1254	CP1254, CP-1254, 1254, MS-TURK
WINDOWS-1255	CP1255, CP-1255, 1255, MS-HEBR
WINDOWS-1256	CP1256, CP-1256, 1256, MS-ARAB
WINDOWS-1257	CP1257, CP-1257, 1257, WINBALTRIM
WINDOWS-1258	CP1258, CP-1258, 1258
WINDOWS-874	CP874, CP-874, 874, IBM874, IBM-874
WINDOWS-31J	WINDOWS-932, CP932, CP-932, 932
WINDOWS-936	CP936, CP-936, 936, GBK, MS936, MS-936
CP949	WINDOWS-949, CP-949, 949
BIG5	WINDOWS-950, CP950, CP-950, 950, BIG, BIG5, BIG-5, BIG-FIVE, BIGFIVE, CN-BIG5, BIG5-CP950
SHIFT-JIS	SJIS
UTF8	UTF-8, ISO10646=UTF8

Appendix E. Customer Service

The Customer Service of Exasol AG supports its customers in all matters relating to installation, commissioning, operation and use of Exasol. Customer Service is pleased to address any concerns of our customers.

Requests

We recommend that all requests are sent to the following email address: <service@exasol.com>. A brief description of the issue by email or, in urgent cases, by phone will suffice. We would ask you to always advise us of your contact information.

Internal processes

Customer Service receives all requests and, if possible, responds immediately. If an immediate reply is not possible, the request will be categorized and passed on to the relevant department at Exasol. In this case, our customer receives feedback on when a conclusive or intermediate answer can be expected.

Downloads

This manual as well as the latest software such as drivers and other information are available on the customer portal on our website wwwexasol.com.

Contact information

Exasol AG
Customer Service
Phone: 00800 EXASUPPORT (00800 3927 877 678)
Email: <service@exasol.com>

Abbreviations

A

ADO.NET	Abstract Data Objects .NET
ANSI	American National Standards Institute
API	Application Programming Interface

B

BI	Business Intelligence
BOM	Byte Order Mark

C

CLI	Call Level Interface
CSV	Comma Separated Values

D

DB	Database
DBA	Database Administrator
DBMS	Database Management System
DCL	Data Control Language
DDL	Data Definition Language
DiagRec	Diagnostic Record, specified in the ODBC standard
DML	Data Manipulation Language
DNS	Domain Name Service
DQL	Data Query Language
DSN	Data Source Name

E

ETL	Extract, Transform, Load
-----	--------------------------

F

FBV	Fix Block Values
-----	------------------

G

GB	$\text{Gigabyte} = 10^9 = 1.000.000.000 \text{ Bytes}$
GiB	$\text{Gibibyte} = 2^{30} = 1.073.741.824 \text{ Bytes}$

H

HPC	High Performance Computing
I	

ISO	International Standards Organization
J	

JDBC	Java DataBase Connectivity
JRE	Java Runtime Environment
JSON	JavaScript Object Notation - an open -standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs
JVM	Java Virtual Machine

K

kB	$\text{Kilobyte} = 10^3 = 1.000 \text{ Bytes}$
kiB	$\text{Kibibyte} = 2^{10} = 1.024 \text{ Bytes}$

L

LDAP	Lightweight Directory Access Protocol (authentication service)
M	

MB	$\text{Megabyte} = 10^6 = 1.000.000 \text{ Bytes}$
MiB	$\text{Mebibyte} = 2^{20} = 1.048.576 \text{ Bytes}$

O

ODBC	Open DataBase Connectivity
OLE DB	Object Linking and Embedding DataBase

P

PCRE	Perl Compatible Regular Expressions
POSIX	Portable Operating System Interface [for Unix]

POSIX BRE	POSIX Basic Regular Expressions
POSIX ERE	POSIX Extended Regular Expressions

S

SASL	Simple Authentication and Security Layer
SDK	Software Development Kit
SQL	Standard Query Language

T

TMS	Transaction Management System
-----	-------------------------------

U

UDF	User Defined Function
-----	-----------------------

W

WKB	Well Known Binary (binary representation of geospatial data)
WKT	Well Known Text (text representation of geospatial data)

Index

A

ABS function, 143
 ACOS function, 143
 ADD_DAYS function, 144
 ADD_HOURS function, 144
 ADD_MINUTES function, 145
 ADD_MONTHS function, 145
 ADD_SECONDS function, 146
 ADD_WEEKS function, 146
 ADD_YEARS function, 147
 ADO.NET Data Destination, 397
 ADO.NET Data Processing Extension, 399
 ADO.NET Data Provider, 393
 Installation, 393
 Use, 393
 Alias, 77
 ALTER ANY CONNECTION system privilege, 35
 ALTER ANY CONNECTION System privilege, 66
 ALTER ANY SCHEMA system privilege, 14
 ALTER ANY TABLE system privilege, 19, 23, 24, 36
 ALTER ANY VIRTUAL SCHEMA REFRESH system privilege, 14
 ALTER ANY VIRTUAL SCHEMA system privilege, 14
 ALTER CONNECTION statement, 66
 ALTER object privilege, 14, 19, 23, 24
 ALTER SCHEMA statement, 14
 ALTER SESSION statement, 91
 ALTER SYSTEM statement, 93
 ALTER TABLE statement
 ADD COLUMN, 19
 ADD CONSTRAINT, 24
 ALTER COLUMN DEFAULT, 19
 ALTER COLUMN IDENTITY, 19
 DISTRIBUTE BY, 23
 DROP COLUMN, 19
 DROP CONSTRAINT, 24
 DROP DISTRIBUTION KEYS, 23
 MODIFY COLUMN, 19
 MODIFY CONSTRAINT, 24
 RENAME COLUMN, 19
 RENAME CONSTRAINT, 24
 ALTER USER statement, 62
 ALTER USER System privilege, 36
 ALTER USER system privilege, 62
 AND predicate, 131
 APPROXIMATE_COUNT_DISTINCT Function, 148
 ASC, 78
 ASCII function, 148
 ASIN function, 149
 Assignment, 28
 ATAN function, 149
 ATAN2 function, 149
 Auditing, 439

Authentification
 LDAP, 62
 Password, 61
 AVG function, 150

B

BETWEEN predicate, 132
 BIT_AND function, 150
 BIT_CHECK function, 151
 BIT_LENGTH function, 152
 BIT_LROTATE function, 152
 BIT_LSHIFT function, 153
 BIT_NOT function, 153
 BIT_OR function, 154
 BIT_RROTATE function, 154
 BIT_RSHIFT function, 155
 BIT_SET function, 155
 BIT_TO_NUM function, 156
 BIT_XOR function, 157
 BOOLEAN data type, 105
 BucketFS, 65, 323

C

CASCADE
 in DROP SCHEMA statement, 13
 in DROP VIEW Statement, 27
 CASCADE CONSTRAINTS
 in ALTER TABLE statement, 21
 in DROP TABLE statement, 19
 in REVOKE statement, 72
 CASE function, 157
 CAST function, 158
 CEIL function, 159
 CEILING function, 159
 CHAR data type, 107
 CHARACTER_LENGTH function, 159
 CH[A]R function, 160
 CLI
 Best Practice, 403
 Example, 402
 General, 400
 Linux/Unix version, 401
 Windows version, 400
 CLOSE SCHEMA statement, 96
 Cluster Enlargement, 100
 COALESCE function, 160
 COLOGNE_PHONETIC function, 161
 Column
 Add column, 19
 Alter column identity, 19
 Change data type, 19
 Comment column, 36
 Delete column, 19
 Rename column, 19
 Set default value, 19
 Column alias, 77
 COMMENT statement, 36

Comments, 5
COMMIT Statement, 88
CONCAT function, 161
CONNECT BY, 77
 CONNECT_BY_ISCYCLE, 78
 CONNECT_BY_ISLEAF, 78
 CONNECT_BY_ROOT, 77
 LEVEL, 77
 NO_CYCLE, 77
 START WITH, 77
 SYS_CONNECT_BY_PATH, 77
Connection
 Alter connection, 66
 Comment connection, 36
 Create connection, 65
 Drop Connection, 66
 Grant connection, 67
 Rename connection, 35
 Revoke connection, 70
CONNECT_BY_ISCYCLE, 78
CONNECT_BY_ISCYCLE function, 162
CONNECT_BY_ISLEAF, 78
CONNECT_BY_ISLEAF function, 162
CONNECT_BY_ROOT, 77
Constraints, 17
 FOREIGN KEY, 25
 NOT NULL, 25
 PRIMARY KEY, 25
 Status
 DISABLE, 25, 93, 95
 ENABLE, 25, 93, 95
CONSTRAINT_STATE_DEFAULT, 25, 93, 95
CONVERT function, 163
CONVERT_TZ function, 164
CORR function, 165
COS function, 165
COSH function, 166
COT function, 166
COUNT Function, 167
COVAR_POP function, 168
COVAR_SAMP function, 168
CREATE ANY FUNCTION system privilege, 27
CREATE ANY SCRIPT system privilege, 30
CREATE ANY TABLE system privilege, 15, 18
CREATE ANY VIEW system privilege, 26
CREATE CONNECTION statement, 65
CREATE CONNECTION system privilege, 36
CREATE CONNECTION System privilege, 65
CREATE FUNCTION statement, 27
CREATE FUNCTION system privilege, 27
CREATE ROLE statement, 63
CREATE ROLE system privilege, 35, 36
CREATE SCHEMA statement, 12
CREATE SCHEMA system privilege, 12
CREATE SCRIPT statement, 30
CREATE SCRIPT system privilege, 30
CREATE TABLE statement, 15
CREATE TABLE system privilege, 15, 18
CREATE USER statement, 61
CREATE USER system privilege, 35, 36, 61
CREATE VIEW statement, 25
CREATE VIEW system privilege, 26
CREATE VIRTUAL SCHEMA system privilege, 12
CROSS JOIN, 76
CSV Data format, 269
CUBE, 78
CURDATE function, 169
CURRENT_DATE function, 169
CURRENT_SCHEMA, 12
CURRENT_SCHEMA function, 170
CURRENT_SESSION function, 170
CURRENT_STATEMENT function, 171
CURRENT_TIMESTAMP function, 171
CURRENT_USER function, 172

D

Data Destination, 393, 397
Data Processing Extension, 393, 399
Data types, 104
 Aliases, 107, 108
Date/Time
 DATE, 105
 INTERVAL DAY TO SECOND, 107
 INTERVAL YEAR TO MONTH, 107
 TIMESTAMP, 105
 TIMESTAMP WITH LOCAL TIME ZONE, 105
Details, 104
GEOMETRY, 107
Numeric
 DECIMAL, 104
 DOUBLE PRECISION, 104
Overview, 104
Strings
 CHAR, 107
 VARCHAR, 107
Type conversion rules, 108
Database
 Reorganize, 100
DATE data type, 105
DATE_TRUNC function, 172
DAY function, 173
DAYS_BETWEEN function, 173
DBTIMEZONE function, 174
DCL statements, 61
 ALTER CONNECTION, 66
 ALTER USER, 62
 CREATE CONNECTION, 65
 CREATE ROLE, 63
 CREATE USER, 61
 DROP CONNECTION, 66
 DROP ROLE, 64
 DROP USER, 63
 GRANT, 67
 REVOKE, 70
DDEX provider, 393

- DDL statements, 12
 ALTER SCHEMA, 14
 ALTER TABLE
 ADD COLUMN, 19
 ALTER COLUMN DEFAULT, 19
 ALTER COLUMN IDENTITY, 19
 DISTRIBUTE BY, 23
 DROP COLUMN, 19
 DROP DISTRIBUTION KEYS, 23
 MODIFY COLUMN, 19
 RENAME COLUMN, 19
 COMMENT, 36
 CREATE FUNCTION, 27
 CREATE SCHEMA, 12
 CREATE SCRIPT, 30
 CREATE TABLE, 15
 CREATE VIEW, 25
 DROP FUNCTION, 30
 DROP SCHEMA, 13
 DROP SCRIPT, 34
 DROP TABLE, 19
 DROP VIEW, 26
 RENAME, 35
 SELECT INTO, 18
- DDL Statements**
- ALTER TABLE
 - ADD CONSTRAINT, 24
 - DROP CONSTRAINT, 24
 - MODIFY CONSTRAINT, 24
 - RENAME CONSTRAINT, 24
- DECIMAL data type, 104
 DECODE function, 174
 Default values for columns, 17, 19, 21, 22, 38, 39, 41, 110
 - Delete default value, 22
 - Display, 111
 - Example, 110, 111
 - Permitted values, 111
 - Possible sources of error, 112
- DEFAULT_LIKE_ESCAPE_CHARACTER, 92, 94
 DEGREES function, 175
 DELETE ANY TABLE system privilege, 43
 DELETE object privilege, 43
 DELETE statement, 42
 DENSE_RANK function, 176
 DESC, 78
 DESCRIBE statement, 97
 DISABLE, 25
 DISTINCT, 77
 DIV function, 176
 DML statements, 38
 - DELETE, 42
 - EXPORT, 52
 - IMPORT, 44
 - INSERT, 38
 - MERGE, 40
 - TRUNCATE, 43
 - UPDATE, 39
- DOUBLE PRECISION data type, 104
 DROP ANY CONNECTION System privilege, 67
 DROP ANY FUNCTION system privilege, 30
 DROP ANY SCHEMA system privilege, 13
 DROP ANY SCRIPT system privilege, 34
 DROP ANY TABLE system privilege, 19
 DROP ANY VIEW system privilege, 27
 DROP ANY VIRTUAL SCHEMA system privilege, 13
 DROP CONNECTION statement, 66
 DROP DEFAULT clause, 22
 DROP FUNCTION statement, 30
 DROP ROLE statement, 64
 DROP SCHEMA statement, 13
 DROP SCRIPT statement, 34
 DROP TABLE statement, 19
 DROP USER statement, 63
 DROP USER system privilege, 63
 DROP VIEW statement, 26
 DUMP function, 177

E

- EDIT_DISTANCE function, 178
 emit(), 302, 307, 314, 319
 EMITS, 295
 ENABLE, 25
 ETL, 265
 - CSV Data format, 269
 - EXPORT command, 52
 - FBV Data format, 271
 - File formats, 269
 - Hadoop support, 268
 - IMPORT command, 44
 - Scripting, 266
 - SQL commands, 265
 - User-defined EXPORT using UDFs, 268
 - User-defined IMPORT using UDFs, 266
 - Virtual schemas, 269
- EXAplus, 353
 - Console mode, 358
 - EXAplus-specific commands, 361
 - Installation, 353
 - User interface, 354
- EXAplus commands, 362
 - @, 362
 - @@, 363
 - ACCEPT, 363
 - BATCH, 363
 - COLUMN, 364
 - CONNECT, 366
 - DEFINE, 366
 - DISCONNECT, 367
 - EXIT, 367
 - HOST, 367
 - PAUSE, 368
 - PROMPT, 368
 - QUIT, 367

SET AUTOCOMMIT, 368
SET AUTOCOMPLETION, 369
SET COLSEPARATOR, 369
SET DEFINE, 369
SET ENCODING, 370
SET ESCAPE, 370
SET FEEDBACK, 371
SET HEADING, 371
SET LINESIZE, 372
SET NULL, 372
SET NUMFORMAT, 372
SET PAGESIZE, 373
SET SPOOL ROW SEPARATOR, 373
SET TIME, 374
SET TIMING, 374
SET TRUNCATE HEADING, 374
SET VERBOSE, 375
SHOW, 375
SPOOL, 376
START, 362
TIMING, 376
UNDEFINE, 377
WHENEVER, 377
EXA_TIME_ZONES, 92, 94, 106, 164
EXCEPT, 80
(see also MINUS)
EXECUTE ANY FUNCTION system privilege, 27
EXECUTE ANY SCRIPT system privilege, 31
EXECUTE ANY SCRIPT System privilege, 89
EXECUTE object privilege, 27
EXECUTE Object privilege, 31, 89
EXECUTE SCRIPT statement, 89
EXISTS predicate, 132
EXP function, 178
EXPLAIN VIRTUAL statement, 98
EXPORT statement, 52
EXTRACT function, 179

F

FBV Data format, 271
FIRST_VALUE function, 179
FLOOR function, 180
FLUSH STATISTICS Statement, 101
FOR loop, 28
FOREIGN KEY, 25
Verification of the property, 85
Format models, 122
Date/Time, 122
Numeric, 124
FROM, 77
FROM_POSIX_TIME Function, 181
FULL OUTER JOIN, 76
Function
 Rename function, 35
Functions, 136
Aggregate functions, 140
 APPROXIMATE_COUNT_DISTINCT, 148
 AVG, 150
 CORR, 165
 COUNT, 167
 COVAR_POP, 168
 COVAR_SAMP, 168
 FIRST_VALUE, 179
 GROUPING[_ID], 183
 GROUP_CONCAT, 182
 LAST_VALUE, 190
 MAX, 199
 MEDIAN, 200
 MIN, 201
 PERCENTILE_CONT, 209
 PERCENTILE_DISC, 210
 REGR_*, 218
 REGR_AVGX, 219
 REGR_AVGY, 219
 REGR_COUNT, 219
 REGR_INTERCEPT, 219
 REGR_R2, 219
 REGR_SLOPE, 219
 REGR_SXX, 219
 REGR_SXY, 219
 REGR_SYY, 219
 STDDEV, 233
 STDDEV_POP, 234
 ST_INTERSECTION, 116
 ST_UNION, 117
 SUM, 236
 VARIANCE, 252
 VAR_POP, 251
 VAR_SAMP, 251
Aggregated functions
 STDDEV_SAMP, 235
Analytical functions, 140
 AVG, 150
 CORR, 165
 COUNT, 167
 COVAR_POP, 168
 COVAR_SAMP, 168
 DENSE_RANK, 176
 FIRST_VALUE, 179
 LAG, 189
 LAST_VALUE, 190
 LEAD, 191
 MAX, 199
 MEDIAN, 200
 MIN, 201
 PERCENTILE_CONT, 209
 PERCENTILE_DISC, 210
 RANK, 215
 RATIO_TO_REPORT, 215
 REGR_*, 218
 REGR_AVGY, 219
 REGR_COUNT, 219
 REGR_INTERCEPT, 219
 REGR_R2, 219
 REGR_SLOPE, 219

REGR_SXX, 219
REGR_SXY, 219
REGR_SYY, 219
ROW_NUMBER, 224
STDDEV, 233
STDDEV_POP, 234
STDDEV_SAMP, 235
SUM, 236
VARIANCE, 252
VAR_POP, 251
VAR_SAMP, 251
Bitwise functions, 138
 BIT_AND, 150
 BIT_CHECK, 151
 BIT_LROTATE, 152
 BIT_LSHIFT, 153
 BIT_NOT, 153
 BIT_OR, 154
 BIT_RROTATE, 154
 BIT_RSHIFT, 155
 BIT_SET, 155
 BIT_TO_NUM, 156
 BIT_XOR, 157
Conversion functions, 139
 CAST, 158
 CONVERT, 163
 IS_BOOLEAN, 188
 IS_DATE, 188
 IS_DSINTERVAL, 188
 IS_NUMBER, 188
 IS_TIMESTAMP, 188
 IS_YMINTERVAL, 188
 NUMTODSINTERVAL, 207
 NUMTOYMINTERVAL, 207
 TO_CHAR (datetime), 240
 TO_CHAR (number), 241
 TO_DATE, 241
 TO_DSINTERVAL, 242
 TO_NUMBER, 243
 TO_TIMESTAMP, 243
 TO_YMINTERVAL, 244
Date/Time functions, 137
 ADD_DAYS, 144
 ADD_HOURS, 144
 ADD_MINUTES, 145
 ADD_MONTHS, 145
 ADD_SECONDS, 146
 ADD_WEEKS, 146
 ADD_YEARS, 147
 CONVERT_TZ, 164
 CURDATE, 169
 CURRENT_DATE, 169
 CURRENT_TIMESTAMP, 171
 DATE_TRUNC, 172
 DAY, 173
 DAYS_BETWEEN, 173
 DBTIMEZONE, 174
 EXTRACT, 179
 FROM_POSIX_TIME, 181
 HOUR, 185
 HOURS_BETWEEN, 186
 LOCALTIMESTAMP, 195
 MINUTE, 202
 MINUTES_BETWEEN, 202
 MONTH, 203
 MONTHS_BETWEEN, 204
 NOW, 204
 NUMTODSINTERVAL, 207
 NUMTOYMINTERVAL, 207
 POSIX_TIME, 212
 ROUND (datetime), 223
 SECOND, 227
 SECONDS_BETWEEN, 228
 SESSIONTIMEZONE, 228
 SYSDATE, 238
 SYSTIMESTAMP, 238
 TO_CHAR (datetime), 240
 TO_DATE, 241
 TO_DSINTERVAL, 242
 TO_TIMESTAMP, 243
 TO_YMINTERVAL, 244
 TRUNC[ATE] (datetime), 246
 WEEK, 253
 YEAR, 253
 YEARS_BETWEEN, 254
Geospatial functions, 138
 ST_*, 231
 ST_AREA, 115
 ST_BOUNDARY, 116
 ST_BUFFER, 116
 ST_CENTROID, 116
 ST_CONTAINS, 116
 ST_CONVEXHULL, 116
 ST_CROSSES, 116
 ST_DIFFERENCE, 116
 ST_DIMENSION, 116
 ST_DISJOINT, 116
 ST_DISTANCE, 116
 ST_ENDPOINT, 115
 ST_ENVELOPE, 116
 ST_EQUALS, 116
 ST_EXTERIORRING, 116
 ST_FORCE2D, 116
 ST_GEOMETRYN, 116
 ST_GEOMETRYTYPE, 116
 ST_INTERIORRINGN, 116
 ST_INTERSECTION, 116
 ST_INTERSECTS, 116
 ST_ISCLOSED, 115
 ST_ISEMPTY, 116
 ST_ISRING, 115
 ST_ISSIMPLE, 116
 ST_LENGTH, 115
 ST_NUMGEOMETRIES, 116
 ST_NUMINTERIORRINGS, 116
 ST_NUMPOINTS, 115

ST_OVERLAPS, 117
ST_POINTN, 115
ST_SETSRID, 117
ST_STARTPOINT, 115
ST_SYMDIFFERENCE, 117
ST_TOUCHES, 117
ST_TRANSFORM, 117
ST_UNION, 117
ST_WITHIN, 117
ST_X, 115
ST_Y, 115

Hierarchical queries, 139
CONNECT_BY_ISCYCLE, 162
CONNECT_BY_ISLEAF, 162
LEVEL, 194
SYS_CONNECT_BY_PATH, 237

Numeric functions, 136
ABS, 143
ACOS, 143
ASIN, 149
ATAN, 149
ATAN2, 149
CEIL, 159
CEILING, 159
COS, 165
COSH, 166
COT, 166
DEGREES, 175
DIV, 176
EXP, 178
FLOOR, 180
LN, 195
LOG, 196
LOG10, 197
LOG2, 197
MOD, 203
PI, 211
POWER, 213
RADIAN, 213
RAND[OM], 214
ROUND (number), 224
SIGN, 229
SIN, 229
SINH, 229
SQRT, 231
TAN, 239
TANH, 239
TO_CHAR (number), 241
TRUNC[ATE] (number), 247

Other scalar functions, 139
CASE, 157
COALESCE, 160
CURRENT_SCHEMA, 170
CURRENT_SESSION, 170
CURRENT_STATEMENT, 171
CURRENT_USER, 172
DECODE, 174
GREATEST, 181

HASH_MD5, 183
HASH_SHA[1], 184
HASH_TIGER, 185
IPROC, 188
LEAST, 193
NPROC, 205
NULLIF, 206
NULLIFZERO, 206
NVL, 208
NVL2, 208
ROWID, 225
SYS_GUID, 237
USER, 249
VALUE2PROC, 250
ZEROIFNULL, 254

String functions, 137
ASCII, 148
BIT_LENGTH, 152
CHARACTER_LENGTH, 159
CH[A]R, 160
COLOGNE_PHONETIC, 161
CONCAT, 161
DUMP, 177
EDIT_DISTANCE, 178
INSERT, 186
INSTR, 187
LCASE, 191
LEFT, 193
LENGTH, 194
LOCATE, 196
LOWER, 198
LPAD, 198
LTRIM, 199
MID, 201
OCTET_LENGTH, 209
POSITION, 212
REGEXP_INSTR, 216
REGEXP_REPLACE, 217
REGEXP_SUBSTR, 218
REPEAT, 220
REPLACE, 221
REVERSE, 221
RIGHT, 222
RPAD, 226
RTRIM, 226
SOUNDEX, 230
SPACE, 230
SUBSTR, 235
SUBSTRING, 235
TRANSLATE, 245
TRIM, 245
UCASE, 247
UNICODE, 248
UNICODECHR, 248
UPPER, 249

User-defined
CREATE FUNCTION, 27
DROP FUNCTION, 30

G

GEOMETRY data type, 107
 GEOMETRY Object, 114
 GEOMETRYCOLLECTION Object, 114
 Geospatial data, 114
 Empty set, 114
 Functions, 115
 ST_AREA, 115
 ST_BOUNDARY, 116
 ST_BUFFER, 116
 ST_CENTROID, 116
 ST_CONTAINS, 116
 ST_CONVEXHULL, 116
 ST_CROSSES, 116
 ST_DIFFERENCE, 116
 ST_DIMENSION, 116
 ST_DISJOINT, 116
 ST_DISTANCE, 116
 ST_ENDPOINT, 115
 ST_ENVELOPE, 116
 ST_EQUALS, 116
 ST_EXTERIORRING, 116
 ST_FORCE2D, 116
 ST_GEOMETRYN, 116
 ST_GEOMETRYTYPE, 116
 ST_INTERIORRINGN, 116
 ST_INTERSECTION, 116
 ST_INTERSECTS, 116
 ST_ISCLOSED, 115
 ST_ISEMPTY, 116
 ST_ISRING, 115
 ST_ISSIMPLE, 116
 ST_LENGTH, 115
 ST_NUMGEOMETRIES, 116
 ST_NUMINTERIORRINGS, 116
 ST_NUMPOINTS, 115
 ST_OVERLAPS, 117
 ST_POINTN, 115
 ST_SETSRID, 117
 ST_STARTPOINT, 115
 ST_SYMDIFFERENCE, 117
 ST_TOUCHES, 117
 ST_TRANSFORM, 117
 ST_UNION, 117
 ST_WITHIN, 117
 ST_X, 115
 ST_Y, 115
 Objects, 114
 Empty set, 114
 GEOMETRY, 114
 GEOMETRYCOLLECTION, 114
 LINEARRING, 114
 LINESTRING, 114
 MULTILINESTRING, 114
 MULTIPOINT, 114
 MULTIPOLYGON, 114
 POINT, 114

POLYGON, 114
 getBigDecimal(), 307
 getBoolean(), 307
 getDate(), 307
 getDouble(), 307
 getInteger(), 307
 getLong(), 307
 getString(), 307
 getTimestamp(), 307
 GRANT ANY CONNECTION system privilege, 67, 70
 GRANT ANY OBJECT PRIVILEGE system privilege, 67, 70
 GRANT ANY PRIORITY system privilege, 67, 70
 GRANT ANY PRIVILEGE system privilege, 67, 69, 70
 GRANT ANY ROLE system privilege, 67, 70
 GRANT statement, 67
 Graph analytics, 77
 Graph search, 77
 GREATEST function, 181
 GROUPING SETS, 78
 GROUPING[_ID] function, 183
 GROUP_CONCAT function, 182

H

Hadoop, 266, 268
 HASH_MD5 function, 183
 HASH_SHA[1] function, 184
 HASH_TIGER function, 185
 HAVING, 78
 HCatalog, 268
 HOUR function, 185
 HOURS_BETWEEN function, 186

I

Identifier, 5
 delimited, 6
 regular, 5
 reserved words, 7
 schema-qualified, 7
 Identity columns, 19, 21, 38, 39, 41, 112
 Display, 113
 Example, 112
 Set and change, 113
 IF branch, 28
 IF EXISTS
 in DROP COLUMN statement, 21
 in DROP CONNECTION statement, 67
 in DROP FUNCTION statement, 30
 in DROP ROLE statement, 64
 in DROP SCHEMA statement, 13
 in DROP SCRIPT statement, 34
 in DROP TABLE statement, 19
 in DROP USER statement, 63
 in DROP VIEW statement, 27
 IMPORT statement, 44, 79
 IN predicate, 133

INNER JOIN, 76
INSERT ANY TABLE system privilege, 38
INSERT function, 186
INSERT object privilege, 38
INSERT statement, 38
INSTR function, 187
Interfaces
 ADO.NET Data Provider, 393
 EXAplus, 353
 JDBC driver, 387
 ODBC driver, 378
 SDK, 400
 WebSockets, 400
INTERSECT, 80
INTERVAL DAY TO SECOND data type, 107
INTERVAL YEAR TO MONTH data type, 107
IPROC function, 188
IS NULL predicate, 134
IS_BOOLEAN function, 188
IS_DATE function, 188
IS_DSINTERVAL function, 188
IS_NUMBER function, 188
IS_TIMESTAMP function, 188
IS_YMINTERVAL function, 188

J

Java, 306
JDBC driver, 387
 Best Practice, 392
 Standards, 387
 System requirements, 387
 Use, 388
Join, 76, 77
 CROSS JOIN, 76
 FULL OUTER JOIN, 76
 INNER JOIN, 76
 LEFT JOIN, 76
 OUTER JOIN, 76
 RIGHT JOIN, 76

K

Kerberos, 61, 384, 391
KILL statement, 90

L

LAG function, 189
LAST_VALUE function, 190
LCASE function, 191
LDAP, 62
LEAD function, 191
LEAST function, 193
LEFT function, 193
LEFT JOIN, 76
LENGTH function, 194
LEVEL, 77
LEVEL function, 194
LIKE predicate, 135

LIMIT, 79
LINEARRING Object, 114
LINESTRING Object, 114
Literals, 118
 Boolean, 119
 Date/Time, 119
 Examples, 118
 Interval, 119
 NULL, 121
 Numeric, 118
 Strings, 121
LN function, 195
LOCAL, 77
LOCALTIMESTAMP function, 195
LOCATE function, 196
LOG function, 196
LOG10 function, 197
LOG2 function, 197
LOWER function, 198
LPAD function, 198
LTRIM function, 199
Lua, 301

M

MapReduce, 300
MAX function, 199
MEDIAN function, 200
MERGE statement, 40
MID function, 201
MIN function, 201
MINUS, 80
MINUTE function, 202
MINUTES_BETWEEN function, 202
MOD function, 203
MONTH function, 203
MONTHS_BETWEEN function, 204
MULTILINESTRING Object, 114
MULTIPOINT Object, 114
MULTIPOLYGON Object, 114

N

next(), 302, 307, 314
next_row(), 318
NICE, 93, 263, 411, 420, 435, 440, 449
NLS_DATE_FORMAT, 92, 94
NLS_DATE_LANGUAGE, 92, 94
NLS_FIRST_DAY_OF_WEEK, 92, 94, 172, 223, 246
NLS_NUMERIC_CHARACTERS, 92, 94
NLS_TIMESTAMP_FORMAT, 92, 94
NO_CYCLE, 77
NOT NULL, 25
NOT predicate, 132
NOW function, 204
NPROC function, 205
NULL
 Literal, 121
NULLIF function, 206

`NULIFZERO` function, 206

`NULLS FIRST`, 79

`NULLS LAST`, 79

`NUMTODSINTERVAL` function, 207

`NUMTOYMINTERVAL` function, 207

`NVL` function, 208

`NVL2` function, 208

O

Object privilege

`ACCESS`, 461

`ALTER`, 14, 19, 461

`DELETE`, 43, 461

`EXECUTE`, 27, 31, 89, 461

`INSERT`, 38, 461

`REFERENCES`, 461

`REFRESH`, 14, 461

`SELECT`, 73, 461

 Summary, 459

`UPDATE`, 39, 461

`OCTET_LENGTH` function, 209

ODBC driver, 378

 Best Practice, 386

 Character Sets, 385

 Connection-Strings, 383

 Linux/Unix version, 381

 Standards, 378

 Windows version, 378

 Configuration, 379

 Connection Pooling, 380

 Installation, 378

 Known problems, 380

 System requirements, 378

OPEN SCHEMA statement, 96

Operators, 127

 Arithmetic, 127

 Concatenation operator, 128

`CONNECT BY` operators, 129

OR predicate, 132

OR REPLACE option

 in `CREATE CONNECTION`, 65

 in `CREATE FUNCTION`, 27

 in `CREATE SCRIPT`, 30

 in `CREATE TABLE`, 15

 in `CREATE VIEW`, 26

ORDER BY, 78

OUTER JOIN, 76

P

Pareto, 349

Password, 61

PERCENTILE_CONT function, 209

PERCENTILE_DISC function, 210

PI function, 211

POINT Object, 114

POLYGON Object, 114

POSITION function, 212

`POSIX_TIME` Function, 212

`POWER` function, 213

Predicates, 130

`BETWEEN`, 132

 Comparisons, 130

`EXISTS`, 132

`IN`, 133

`IS NULL`, 134

`LIKE`, 135

 Logical join, 131

`AND`, 131

`NOT`, 132

`OR`, 132

 Precedence of predicates, 130

`REGEXP_LIKE`, 134

`PREFERRING`, 40, 43, 78, 349, 350

`PRELOAD` statement, 102

Preprocessor, 336

PRIMARY KEY, 25

 Verification of the property, 83

PRIOR, 77

Priorities, 263, 410, 411, 412, 419, 420, 421, 426, 435, 436, 440, 449

 Example, 264

 Introduction, 263

 NICE, 93, 263

 Priorities in Exasol, 263

Priority

 Grant priority, 67

 Revoke priority, 70

PROFILE, 93, 95

Profiling, 346, 440, 441, 454, 455

 Activation and Analyzing, 346

 Example, 347

 Introduction, 346

 System parameter, 93, 95

Protocol Buffers, 328

Python, 313

Q

Query Cache, 92, 94

Query Cache for SQL queries, 92, 94

Query Timeout, 92, 95

QUERY_CACHE, 92, 94

QUERY_TIMEOUT, 92, 95

R

R (Programming language), 317

RADIANS function, 213

RAND[OM] function, 214

RANK function, 215

RATIO_TO_REPORT function, 215

RECOMPRESS statement, 99

REFRESH object privilege, 14

REGEXP_INSTR function, 216

REGEXP_LIKE predicate, 134

REGEXP_REPLACE function, 217

REGEXP_SUBSTR function, 218
REGR_* functions, 218
REGR_AVGX function, 219
REGR_AVGY function, 219
REGR_COUNT function, 219
REGR_INTERCEPT function, 219
REGR_R2 function, 219
REGR_SLOPE function, 219
REGR_SXX function, 219
REGR_SXY function, 219
REGR_SYY function, 219
Regular expressions, 8
 Examples, 8
 Pattern elements, 8
RENAME statement, 35
REORGANIZE statement, 100
REPEAT function, 220
REPLACE function, 221
Reserved words, 7
reset(), 302, 307, 314, 318
RESTRICT
 in DROP SCHEMA statement, 13
 in DROP VIEW statement, 27
RETURNS, 295
REVERSE function, 221
REVOKE statement, 70
RIGHT function, 222
RIGHT JOIN, 76
Rights management, 259
 Access control with SQL statements, 260
 Example, 261
 Meta information, 261
 Privileges, 260
 Roles, 259
 User, 259
Roles
 Create roles, 63
 Delete roles, 64
 Grant rights, 67
 Revoke rights, 70
ROLLBACK statement, 89
ROLLUP, 78
ROUND (datetime) function, 223
ROUND (number) function, 224
ROWID, 225
ROW_NUMBER function, 224
RPAD function, 226
RTRIM function, 226

S

SCALAR, 295
Schema
 change owner, 14
 close, 96
 Comment schema, 36
 create, 12
 delete, 13
 open, 96
 rename, 35
Schema objects, 12
Schema-qualified identifiers, 7
Script
 Comment script, 36
 Creating a script, 30
 Dropping a script, 34
 Executing a script, 89
Functions
 exit(), 284
 import(), 284
 output(), 286
 Rename script, 35
Scripting, 272
 Arrays, 275
 Auxiliary functions for identifiers, 286
 Comments, 273
 Control structures, 276
 Debug output, 286
 Dictionary Tables, 275
 Error Handling, 279
 Example, 272
 Executing SQL statements, 280
 Execution blocks, 276
 Functions, 279
 error(), 279
 join(), 287
 pairs(), 275
 pcall(), 279
 pquery(), 280
 query(), 280
 quote(), 287
 sqlparsing.find(), 338
 sqlparsing.getSqltext(), 339
 sqlparsing.isany(), 338
 sqlparsing.iscomment(), 337
 sqlparsing.isidentifier(), 337
 sqlparsing.iskeyword(), 337
 sqlparsing.isnumericliteral(), 338
 sqlparsing.isstringliteral(), 338
 sqlparsing.iswhitespace(), 337
 sqlparsing.iswhitespaceorcomment(), 337
 sqlparsing.normalize(), 338
 sqlparsing.setSqltext(), 339
 sqlparsing.tokenize(), 337
 string.find(), 288
 string.format(formatstring, e1, e2, ...), 290
 string.gmatch(s, pattern), 288
 string.gsub(s, pattern, repl [, n]), 288
 string.len(s), 289
 string.lower(s), 289
 string.match(s, pattern [, init]), 288
 string.rep(s, n), 289
 string.reverse(s), 289
 string.sub(s, i [, j]), 288
 string.upper(s), 289
 table.concat(), 294

table.insert(), 293
table.maxn(), 294
table.remove(), 293
table.sort(), 294
Further notes, 287
Importing external scripts, 284
Internet access, 292
Language, 272
Lexical conventions, 273
Math library, 292
Metadata, 285
nil, 274
Operators, 278
Parametrized SQL commands, 281
Return value of a script, 284
Script parameters, 283
Simple Variables, 275
SQL parsing, 292
String library, 287
System tables, 294
Table library, 293
Types & Values, 274
Unicode, 291
XML parsing, 291
SCRIPT_LANGUAGES, 93, 95
SDK, 400
 CLI, 400
SECOND function, 227
SECONDS_BETWEEN function, 228
SELECT, 73
 SELECT ANY TABLE system privilege, 73
SELECT INTO statement, 18
SELECT object privilege, 73
SESSIONTIMEZONE function, 228
SET, 295
SIGN function, 229
SIN function, 229
SINH function, 229
size(), 302, 307, 314, 318
Skyline, 349, 350
 Example, 350
 How Skyline works, 349
 Motivation, 349
 Syntax elements, 350
SOUNDEX function, 230
SPACE function, 230
SQL Preprocessor, 336
 Best Practice, 339
 Examples, 339
 Functions
 sqlparsing.find(), 338
 sqlparsing.getsqltext(), 339
 sqlparsing.isany(), 338
 sqlparsing.iscomment(), 337
 sqlparsing.isidentifier(), 337
 sqlparsing.iskeyword(), 337
 sqlparsing.isnumericliteral(), 338
 sqlparsing.isstringliteral(), 338
sqlparsing.iswhitespace(), 337
sqlparsing.iswhitespaceorcomment(), 337
sqlparsing.normalize(), 338
sqlparsing.setsqltext(), 339
sqlparsing.tokenize(), 337
Introduction, 336
 Library sqlparsing, 336
SQL standard, 467
SQL statement, 12
SQL_PREPROCESSOR_SCRIPT, 93, 95
SQRT function, 231
START WITH, 77
STDDEV function, 233
STDDEV_POP function, 234
STDDEV_SAMP function, 235
ST_* functions, 231
ST_AREA Function, 115
ST_BOUNDARY Function, 116
ST_BUFFER Function, 116
ST_CENTROID Function, 116
ST_CONTAINS Function, 116
ST_CONVEXHULL Function, 116
ST_CROSSES Function, 116
ST_DIFFERENCE Function, 116
ST_DIMENSION Function, 116
ST_DISJOINT Function, 116
ST_DISTANCE Function, 116
ST_ENDPOINT Function, 115
ST_ENVELOPE Function, 116
ST_EQUALS Function, 116
ST_EXTERIORRING Function, 116
ST_FORCE2D Function, 116
ST_GEOMETRYN Function, 116
ST_GEOMETRYTYPE Function, 116
ST_INTERIORRINGN Function, 116
ST_INTERSECTION Function, 116
ST_INTERSECTS Function, 116
ST_ISCLOSED Function, 115
ST_ISEMPTY Function, 116
ST_ISRING Function, 115
ST_ISSIMPLE Function, 116
ST_LENGTH Function, 115
ST_NUMGEOMETRIES Function, 116
ST_NUMINTERIORRINGS Function, 116
ST_NUMPOINTS Function, 115
ST_OVERLAPS Function, 117
ST_POINTN Function, 115
ST_SETSRID Function, 117
ST_STARTPOINT Function, 115
ST_SYMDIFFERENCE Function, 117
ST_TOUCHES Function, 117
ST_TRANSFORM Function, 117
ST_UNION Function, 117
ST_WITHIN Function, 117
ST_X Function, 115
ST_Y Function, 115
SUBSTR function, 235
SUBSTRING function, 235

SUM function, 236
SYSDATE function, 238
System parameter
 CONSTRAINT_STATE_DEFAULT, 93, 95
 DEFAULT_LIKE_ESCAPE_CHARACTER, 92, 94
 NICE, 93
 NLS_DATE_FORMAT, 92, 94
 NLS_DATE_LANGUAGE, 92, 94
 NLS_FIRST_DAY_OF_WEEK, 92, 94
 NLS_NUMERIC_CHARACTERS, 92, 94
 NLS_TIMESTAMP_FORMAT, 92, 94
 PROFILE, 93, 95
 QUERY_CACHE, 92, 94
 QUERY_TIMEOUT, 92, 95
 SCRIPT_LANGUAGES, 93, 95
 SQL_PREPROCESSOR_SCRIPT, 93, 95
 TIMESTAMP_ARITHMETIC_BEHAVIOR, 92, 94
 TIME_ZONE, 92, 94

System privilege
 ACCESS ANY CONNECTION, 460
 ALTER ANY CONNECTION, 35, 66, 460
 ALTER ANY SCHEMA, 14, 460
 ALTER ANY TABLE, 19, 460
 ALTER ANY VIRTUAL SCHEMA, 14, 460
 ALTER ANY VIRTUAL SCHEMA REFRESH, 14, 460
 ALTER SYSTEM, 460
 ALTER USER, 62, 460
 CREATE ANY FUNCTION, 27, 461
 CREATE ANY SCRIPT, 30, 461
 CREATE ANY TABLE, 15, 18, 460
 CREATE ANY VIEW, 461
 CREATE CONNECTION, 65, 460
 CREATE FUNCTION, 27, 461
 CREATE ROLE, 35, 460
 CREATE SCHEMA, 12, 460
 CREATE SCRIPT, 30, 461
 CREATE SESSION, 61, 460
 CREATE TABLE, 15, 18, 460
 CREATE USER, 35, 61, 460
 CREATE VIEW, 461
 CREATE VIRTUAL SCHEMA, 12, 460
 DELETE ANY TABLE, 43, 460
 DROP ANY CONNECTION, 460
 DROP ANY FUNCTION, 461
 DROP ANY ROLE, 460
 DROP ANY SCHEMA, 13, 460
 DROP ANY SCRIPT, 461
 DROP ANY TABLE, 19, 460
 DROP ANY VIEW, 461
 DROP ANY VIRTUAL SCHEMA, 13, 460
 DROP USER, 63, 460
 EXECUTE ANY FUNCTION, 27, 461
 EXECUTE ANY SCRIPT, 31, 89, 461
 GRANT ANY CONNECTION, 67, 70, 460
 GRANT ANY OBJECT PRIVILEGE, 67, 70, 460
 GRANT ANY PRIORITY, 67, 70, 460
 GRANT ANY PRIVILEGE, 67, 69, 70, 460

GRANT ANY ROLE, 67, 70, 460
INSERT ANY TABLE, 38, 460
KILL ANY SESSION, 460
SELECT ANY DICTIONARY, 461
SELECT ANY TABLE, 73, 461
Summary, 459
UPDATE ANY TABLE, 39, 461
USE ANY CONNECTION, 44, 52, 460

System tables, 405
 CAT, 457
 DUAL, 457
 EXA_ALL_COLUMNS, 405
 EXA_ALL_CONNECTIONS, 406
 EXA_ALL_CONSTRAINTS, 406
 EXA_ALL_CONSTRAINT_COLUMNS, 407
 EXA_ALL_DEPENDENCIES, 407
 EXA_ALL_FUNCTIONS, 407
 EXA_ALL_INDICES, 408
 EXA_ALL_OBJECTS, 409
 EXA_ALL_OBJECT_SIZES, 410
 EXA_ALL_OBJ_PRIVS, 408
 EXA_ALL_OBJ_PRIVS_MADE, 408
 EXA_ALL_OBJ_PRIVS_REC'D, 409
 EXA_ALL_ROLES, 410
 EXA_ALL_SCRIPTS, 410
 EXA_ALL_SESSIONS, 411
 EXA_ALL_TABLES, 411
 EXA_ALL_USERS, 412
 EXA_ALL_VIEWS, 412
 EXA_ALL_VIRTUAL_COLUMNS, 412
 EXA_ALL_VIRTUAL_SCHEMA_PROPERTIES, 413
 EXA_ALL_VIRTUAL_TABLES, 413
 EXA_DBA_COLUMNS, 413
 EXA_DBA_CONNECTIONS, 414
 EXA_DBA_CONNECTION_PRIVS, 414
 EXA_DBA_CONSTRAINTS, 415
 EXA_DBA_CONSTRAINT_COLUMNS, 415
 EXA_DBA_DEPENDENCIES, 416
 EXA_DBA_DEPENDENCIES_RECURSIVE, 416
 EXA_DBA_FUNCTIONS, 417
 EXA_DBA_INDICES, 417
 EXA_DBA_OBJECTS, 418
 EXA_DBA_OBJECT_SIZES, 418
 EXA_DBA_OBJ_PRIVS, 417
 EXA_DBA_RESTRICTED_OBJ_PRIVS, 415
 EXA_DBA_ROLES, 419
 EXA_DBA_ROLE_PRIVS, 419
 EXA_DBA_SCRIPTS, 419
 EXA_DBA_SESSIONS, 420
 EXA_DBA_SYS_PRIVS, 420
 EXA_DBA_TABLES, 421
 EXA_DBA_USERS, 421
 EXA_DBA_VIEWS, 421
 EXA_DBA_VIRTUAL_COLUMNS, 422
 EXA_DBA_VIRTUAL_SCHEMA_PROPERTIES, 422
 EXA_DBA_VIRTUAL_TABLES, 422

- EXA_LOADAVG, 423
- EXA_METADATA, 423
- EXA_PARAMETERS, 423
- EXA_ROLE_CONNECTION_PRIVS, 423
- EXA_ROLE_OBJ_PRIVS, 424
- EXA_ROLE_RESTRICTED_OBJ_PRIVS, 424
- EXA_ROLE_ROLE_PRIVS, 424
- EXA_ROLE_SYS_PRIVS, 425
- EXA_SCHEMAS, 425
- EXA_SCHEMA_OBJECTS, 425
- EXA_SESSION_CONNECTIONS, 425
- EXA_SESSION_PRIVS, 426
- EXA_SESSION_ROLES, 426
- EXA_SPATIAL_REF_SYS, 426
- EXA_SQL_KEYWORDS, 426
- EXA_SQL_TYPES, 427
- EXA_STATISTICS_OBJECT_SIZES, 427
- EXA_SYSCAT, 405
- EXA_TIME_ZONES, 428
- EXA_USER_COLUMNS, 428
- EXA_USER_CONNECTION_PRIVS, 429
- EXA_USER_CONSTRAINTS, 429
- EXA_USER_CONSTRAINT_COLUMNS, 430
- EXA_USER_DEPENDENCIES, 430
- EXA_USER_FUNCTIONS, 431
- EXA_USER_INDICES, 431
- EXA_USER_OBJECTS, 432
- EXA_USER_OBJECT_SIZES, 433
- EXA_USER_OBJ_PRIVS, 431
- EXA_USER_OBJ_PRIVS_MADE, 432
- EXA_USER_OBJ_PRIVS_REC'D, 432
- EXA_USER_RESTRICTED_OBJ_PRIVS, 429
- EXA_USER_ROLE_PRIVS, 433
- EXA_USER_SCRIPTS, 433
- EXA_USER_SESSIONS, 434
- EXA_USER_SYS_PRIVS, 435
- EXA_USER_TABLES, 435
- EXA_USER_USERS, 435
- EXA_USER_VIEWS, 436
- EXA_USER_VIRTUAL_COLUMNS, 436
- EXA_USER_VIRTUAL_SCHEMA_PROPERTIES, 436
- EXA_USER_VIRTUAL_TABLES, 437
- EXA_VIRTUAL_SCHEMAS, 437
- EXA_VOLUME_USAGE, 437
- Statistics
 - EXA_DB_AUDIT_SESSIONS, 439
 - EXA_DB_AUDIT_SQL, 439
 - EXA_DB_PROFILE_LAST_DAY, 440
 - EXA_DB_PROFILE_RUNNING, 441
 - EXA_DB_SESSIONS_LAST_DAY, 442
 - EXA_DB_TRANSACTION_CONFLICTS, 443
 - EXA_DB_SIZE_DAILY, 444
 - EXA_DB_SIZE_HOURLY, 444
 - EXA_DB_SIZE_LAST_DAY, 443
 - EXA_DB_SIZE_MONTHLY, 445
 - EXA_MONITOR_DAILY, 447
 - EXA_MONITOR_HOURLY, 446
 - EXA_MONITOR_LAST_DAY, 446
 - EXA_MONITOR_MONTHLY, 447
 - EXA_SQL_DAILY, 450
 - EXA_SQL_HOURLY, 449
 - EXA_SQL_LAST_DAY, 448
 - EXA_SQL_MONTHLY, 451
 - EXA_SYSTEM_EVENTS, 452
 - EXA_USAGE_DAILY, 453
 - EXA_USAGE_HOURLY, 453
 - EXA_USAGE_LAST_DAY, 453
 - EXA_USAGE_MONTHLY, 453
 - EXA_USER_PROFILE_LAST_DAY, 454
 - EXA_USER_PROFILE_RUNNING, 455
 - EXA_USER_SESSIONS_LAST_DAY, 455
 - EXA_USER_TRANSACTION_CONFLICTS_LAST_DAY, 456
- System-Parameter
 - TIME_ZONE_BEHAVIOR, 92, 94
- SYSTIMESTAMP function, 238
- SYS_CONNECT_BY_PATH, 77
- SYS_CONNECT_BY_PATH function, 237
- SYS_GUID function, 237

T

Table

- Add column, 19
- Alter column identity, 19
- Change data type, 19
- Change values, 39
- Comment table, 36
- Create table, 15, 18
- Delete all rows, 43
- Delete column, 19
- Delete rows, 42
- Delete table, 19
- Display types, 97
- Export, 52
- Import, 44
- Import change data, 40
- Insert rows, 38
- Loading, 44, 52
- Recompress, 99
- Rename column, 19
- Rename table, 35
- SELECT, 73
- Set default value, 19
- Table operators, 80
- Table operators, 80
- TAN function, 239
- TANH function, 239
- Time zones, 164
- Timeout for queries, 92, 95
- TIMESTAMP data type, 105
- TIMESTAMP WITH LOCAL TIME ZONE data type, 105
- TIMESTAMP_ARITHMETIC_BEHAVIOR, 92, 94
- TIME_ZONE, 92, 94

TIME_ZONE_BEHAVIOR, 92, 94, 105
TO_CHAR (datetime) function, 240
TO_CHAR (number) function, 241
TO_DATE function, 241
TO_DSINTERVAL function, 242
TO_NUMBER function, 243
TO_TIMESTAMP function, 243
TO_YMINTERVAL function, 244
Transaction, 88, 89, 102, 257, 261, 439
Transaction conflict, 428, 439, 443, 456
Transaction management, 257
TRANSLATE function, 245
TRIM function, 245
TRUNCATE AUDIT LOGS Statement, 101
TRUNCATE statement, 43
TRUNC[ATE] (datetime) function, 246
TRUNC[ATE] (number) function, 247

U

UCASE function, 247
UDF scripts, 295
Access to external services, 301
Aggregate and analytical functions, 297
BucketFS, 323
cleanup(), 301, 307, 313, 317
Dynamic input and output parameters, 298
Dynamic parameter list, 302, 308, 313, 317
emit(), 302, 307, 314, 319
EMITS, 295
init(), 307
Introducing examples, 296
Introduction, 295
Java, 306
Lua, 301
MapReduce programs, 300
Metadata, 308
next(), 302, 307, 314
next_row(), 318
ORDER BY, 296
Parameter, 308
Parameters, 301, 313, 317
Performance, 296
Python, 313
R, 317
reset(), 302, 307, 314, 318
RETURNS, 295
run(), 301, 307, 313, 317
SCALAR, 295
Scalar functions, 296
SET, 295
size(), 302, 307, 314, 318
User-defined ETL using UDFs, 301
UNICODE function, 248
UNICODECHR function, 248
UNION [ALL], 80
UNIQUE
Verification of the property, 84

UPDATE ANY TABLE system privilege, 39
UPDATE object privilege, 39
UPDATE statement, 39
UPPER function, 249
User
Change password, 62
Create user, 61
Delete user, 63
Grant rights, 67
Revoke rights, 70
USER function, 249
User-defined functions
Assignment, 28
CREATE FUNCTION, 27
DROP FUNCTION, 30
FOR loop, 28
IF branch, 28
Syntax, 27
WHILE loop, 28
USING, 77

V

VALUE2PROC function, 250
VARCHAR data type, 107
VARIANCE function, 252
VAR_POP function, 251
VAR_SAMP function, 251
View
Comment view, 36
Create view, 25
Delete view, 26
INVALID, 26
Rename view, 35
Status, 26
Virtual schemas, 330
Access concept, 332
Adapters and properties, 331
Details for experts, 334
EXPLAIN VIRTUAL, 98
Metadata, 333
Privileges for administration, 332
Virtual schemas and tables, 330

W

WebSockets, 400
WEEK function, 253
WHERE, 77
WHILE loop, 28
WITH, 76

Y

YEAR function, 253
YEARS_BETWEEN function, 254

Z

ZEROIFNULL function, 254
ZeroMQ, 328