

Database Management System Digital Q.DataBase Version 17.4-251010

Diasoft

2025

This document is the result of intellectual activity, the exclusive right to which belongs to "Diasoft".

Any use (in whole or in part) of this document (in particular: copying, reproduction, distribution, making available to the public, etc., in digital form and/or on paper) is permitted only by agreement with the copyright holder.

The copyright holder reserves the right to make changes to the Software Product and this documentation without prior notice to the Licensee.

Table of Contents

1. General Provisions	7
1.1. What is Digital Q.DataBase?	7
1.2. Main Functional Capabilities	7
1.2.1. Main Advantages	7
1.2.2. Architectural Features	9
1.2.3. Components and Modules	11
2. Terms and Abbreviations	12
3. Installation and Configuration of the DBMS	13
3.1. Supported Operating Systems	13
3.2. System Requirements	13
3.2.1. Minimum System Requirements	13
3.2.2. Recommended System Requirements	13
3.3. Installation Process	14
3.3.1. Installation on Ubuntu OS	15
3.4. DBMS Installation Result	15
3.5. File Structure	16
3.5.1. Executable Files	16
3.5.2. Database Directory	17
3.5.3. Client Part Directory	21
3.6. Digital Q.DataBase Services	22
3.6.1. Service qdatabase-17	22
3.6.2. Service qdbparse	23
3.6.3. Service qdbproxy	23
3.6.4. Service qdbdgrid	24
4. Installation and Configuration of patroni	25
5. Determining Q.DataBase Version	31
6. License	32
6.1. Warnings about License Expiration	32
6.2. Updating the License File	32
7. SQL Dialect Features	33
7.1. Using the Oracle Database Dialect in Digital Q.DataBase	33

7.2. Using the MS SQL Server Dialect in Digital Q.DataBase	34
7.3. Using the PostgreSQL Dialect in Digital Q.DataBase	37
8. Data Types in Digital Q.DataBase	39
9. Working with Digital Q.DataBase using the qsql client application	50
9.1. Creating a Database	50
9.2. Creating Database Objects	51
10. Description of Database Objects	54
10.1. Creating Tables	54
10.2. Creating Indexes	55
10.3. Foreign Keys	56
10.4. Uniqueness Constraints	57
10.5. Additional Database Objects	59
11. Data Modification	61
11.1. Adding Data	61
11.2. Modifying Data	62
11.3. Selecting Data	63
11.4. Deleting Data	64
11.5. Additional Data Modification Operations	65
12. Grouping and Sorting Data	67
12.1. WHERE Clause	67
12.1.1. Logical Operators	67
12.2. GROUP BY Clause	68
12.2.1. Aggregate Functions	69
12.3. ORDER BY Clause	70
12.4. Additional Data Processing Capabilities	70
13. Transactions	73
13.1. Transaction Isolation	74
14. SQL Query Usage Examples	76
14.1. Simple Queries	76
14.2. Complex Data Operations	81
15. Built-in Packages, Functions, and Procedures of the Oracle Database Dialect	90
15.1. General Information	90
15.2. Description of Package Objects	97

15.2.1. DBMS_OUTPUT Package	97
15.2.2. DBMS_ASSERT Package	101
15.2.3. DBMS_RANDOM Package	105
15.2.4. UTL_CALL_STACK Package	111
15.2.5. UTL_ENCODE Package	117
15.2.6. UTL_I18N Package	122
15.2.7. UTL_IDENT Package	135
15.2.8. UTL_INNADR Package	139
15.2.9. UTL_LMS Package	141
15.2.10. UTL_MATCH Package	142
15.2.11. UTL_RAW Package	146
15.2.12. UTL_SMTP Package	162
15.2.13. UTL_TCP Package	176
15.2.14. UTL_MAIL Package	186
15.2.15. UTL_URL Package	191
15.2.16. PLVDATE Package	193
15.2.17. PLVSTR Package	218
15.3. Description of Functions and Procedures	237
16. Built-in Procedures and Functions of MS SQL Server Dialect	256
16.1. General Information	256
16.2. Procedure Descriptions	263
16.3. Function Descriptions	317
17. Built-in functions of MS SQL Server and Oracle Database dialects in PostgreSQL dialect	450
18. Working with the DBMS through external applications	462
18.1. SQL Server Management Studio	462
19. In-memory component	468
19.1. BCP (fast data loading)	474
19.2. dblink	475
20. Administration and Security	505
20.1. User and Access Rights Management	505
20.2. Backup and Recovery	509
20.3. Security and Privacy	511

21. Support Service	513
21.1. Support Service Contacts	513
21.2. Contacting Support Service	513
22. Working with Data Migration Wizard	516
23. Appendix 1. Extensions Included in Q.DataBase	518

1. General Provisions

1.1. What is Digital Q.DataBase?

The Q.DataBase Database Management System (DBMS) is an object-relational database management system (hereinafter referred to as DBMS), developed using the freely distributed PostgreSQL DBMS. The Q.DataBase DBMS is designed to work with various systems, as well as with the banking product line, it provides a significant performance increase.

Digital Q.DataBase is a modern database management system created by the company "Diasoft". This DBMS has unique capabilities aimed at facilitating data migration and compatibility with various SQL dialects. It is built on the basis of the freely distributed PostgreSQL DBMS.

The main advantage of Digital Q.DataBase lies in its ability to automatically recognize and process queries written in dialects of popular DBMSs:

- Oracle Database
- MS SQL Server
- PostgreSQL

Supported Operating Systems

Digital Q.DataBase is available for installation and operation for: Ubuntu 22.04, Ubuntu 24.04

1.2. Main Functional Capabilities

1.2.1. Main Advantages

The Digital Q.DataBase DBMS is developed taking into account the needs of the modern information environment and aims to simplify work with databases. The main advantages of the DBMS are:

- **Compatibility with Different SQL Dialects:** Processing queries written in dialects of popular DBMSs (Oracle Database, MS SQL Server, PostgreSQL), which facilitates data transfer and eliminates the need for query modification.
- **Multimodality:** The DBMS reduces customer infrastructure costs, speeding up application development through simultaneous support for relational tables, JSON

documents, graphs, Key-Value pairs, time series, and columnar analytics at a single core level.

- **Administration via Web Interface:** Convenient remote system management from any device through a browser, simplifying maintenance and minimizing administration costs.
- **Partitioning:** Improving performance and scalability by splitting large tables into separate segments, reducing system load and speeding up data processing.
- **Logical Replication:** Ensuring data consistency between multiple servers through asynchronous distribution of changes, increasing fault tolerance and performance in a distributed environment.
- **Advanced Indexing System:** High query processing speed.
- **Extensibility and Functionality:** Support for DBMS packages and extended stored procedures, allowing for solving complex data processing tasks.
- **Compatibility with SQL Server Management Studio:** Allows efficient management and maintenance of the database using the familiar interface of the popular Microsoft utility, ensuring a smooth transition and reducing staff retraining costs.
- **Database Migration Wizard:** A simple and effective tool for smooth migration of databases from other DBMSs to Digital Q.DataBase.

The Q.DataBase DBMS supports basic PostgreSQL functionality, as well as unique features:

- Loading data obtained by export from MSSQL using the BCP utility;
- 64-bit transaction counter;
- Query plan management using hints written as simple descriptions in special SQL comments;
- Connection pooler, allowing a large number of clients to connect to the DBMS without performance degradation;
- Enhanced basic functionality;
- External extensions (modules):
- `pg_hint_plan`, a module for adjusting execution plans using so-called "hints", written as simple descriptions in special SQL comments;
- `pg_curl` provides the ability to send emails;

- dblink, a module that provides connections to other databases from a database session.

The Q.DataBase DBMS provides performance with the following parameters:

- Number of user sessions supported concurrently – from 1 user session;
- Number of standard queries processed per unit of time – 10,000 queries per second;
- Number of transactions per unit of time – 12,000 transactions per second;
- Delay in execution of a standard DBMS query – 1 millisecond;
- Number of DBMS instances that can work together in load balancing mode – from 2 DBMS instances.

Parameters determining the ability of the DBMS to function in a cluster:

- Maximum number of DBMS instances or their components whose failure leaves the DBMS cluster operational – when 50% of nodes in a 4-node cluster fail, cluster operability is maintained;
- Number of DBMS instances or their components assembled in a cluster – from 2 DBMS instances.

1.2.2. Architectural Features

Let's consider the operation mechanism of Digital Q.DataBase (Fig. 1).

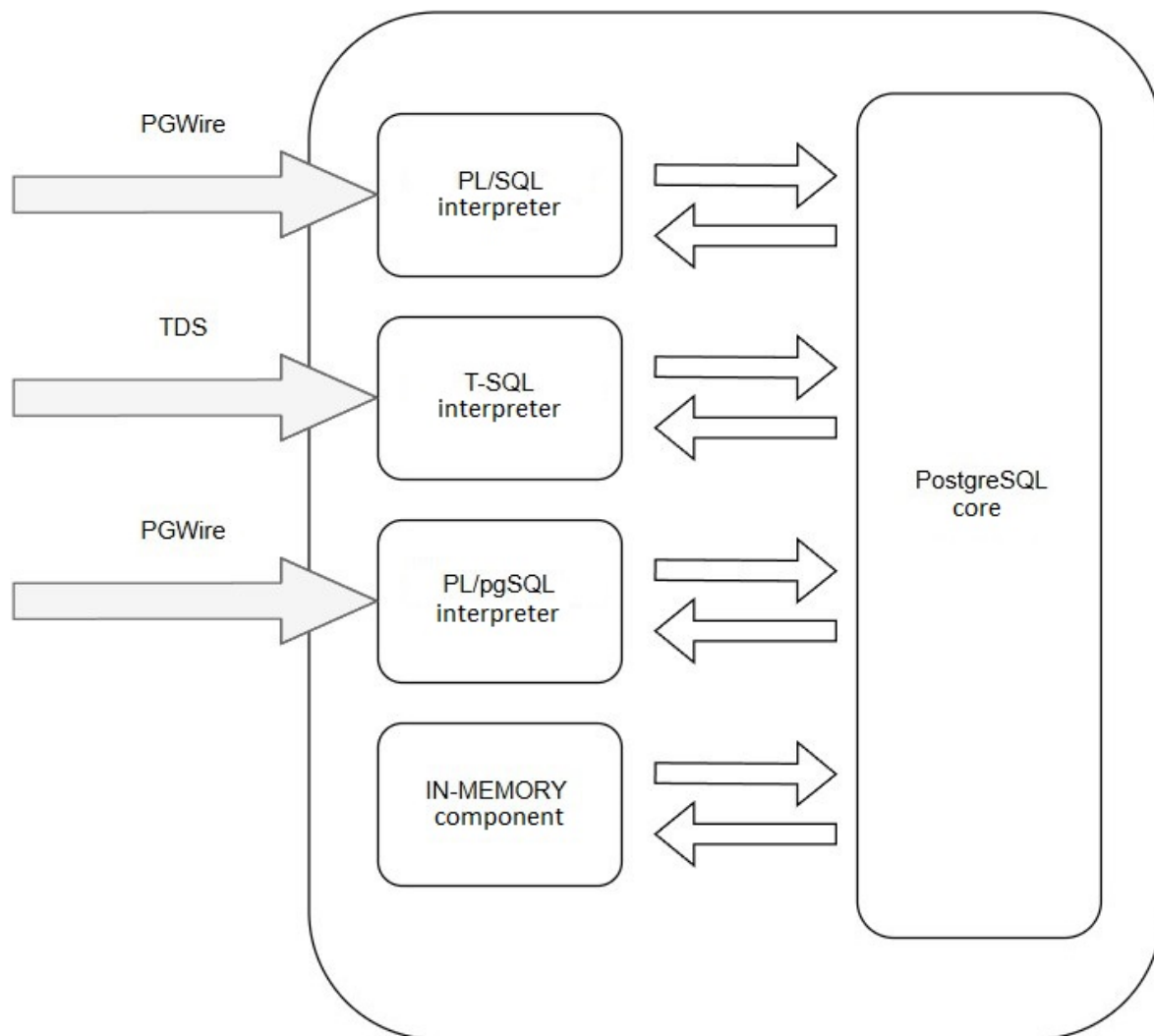


Figure 1. Architectural Diagram of Digital Q.DataBase

The process of interaction with the Digital Q.DataBase database can be described as follows:

1. **Client Request:** A client (application or user) sends a request to perform a specific operation through a specific DBMS port (e.g., data selection, insertion, update, or deletion).
2. **Interaction Protocols:** The request arrives via one of the supported protocols:
 - **PGWire:** Used for interaction with the PostgreSQL and Oracle Database dialects.
 - **TDS:** Used for interaction with the MS SQL Server dialect.

3. **Interpreters:** The request is processed by the corresponding interpreter:
 - **PL/SQL:** Processes queries in the procedural SQL language.
 - **T-SQL:** Processes queries in the Transact-SQL language.
 - **PL/pgSQL:** Processes queries in the PostgreSQL procedural language.
4. **PostgreSQL Core:** The interpreter passes the query to the PostgreSQL core, which performs the following actions:
 - Parses the query.
 - Plans query execution.
 - Executes the query, interacting with data in the database.
5. **IN-MEMORY Component:** If the query requires working with data in RAM, the PostgreSQL core interacts with the IN-MEMORY component (details in section "11. in-memory component").
6. **Response to Client:** After executing the query, the PostgreSQL core returns the result back through the corresponding protocol (TDS, PGWire).

1.2.3. Components and Modules

The DBMS includes the following modules: 1. Digital Q.DataBase Server 2. Command-line tool for interacting with the server 3. Data Migration Wizard

This document contains a description of the operation and implementation of the system's main functionality.

2. Terms and Abbreviations

Term/Abbreviation	Definition
DB	Database
Q.DataBase DBMS	Q.DataBase Database Management System
GUC	Grand Unified Configuration

3. Installation and Configuration of the DBMS

3.1. Supported Operating Systems

Distributions (server and client) are released for the following OS:

- Ubuntu – version 22.04
- Ubuntu – version 24.04

3.2. System Requirements

3.2.1. Minimum System Requirements

Processor

- Multi-threading/CPU cores: minimum 2 cores, better — 4+ cores.
- CPU clock speed: from 2 GHz and above.

RAM

- Minimum: 8 GB RAM.
- Optimal: from 16 GB RAM and above.

Storage (HDD/SDD)

- The disk subsystem must support high input/output (I/O) performance.
- Recommended storage volume depends on data volume and workloads, but a minimum of about 50–100 GB of free space.
- For large data volumes, it is desirable to use SSD disks or RAID arrays to improve read-write operation performance.

3.2.2. Recommended System Requirements

Processor

- Number of cores: 8+ (more is desired, especially for scalable solutions).
- CPU clock speed: from 2.5 GHz and above.
- Hyper-Threading, support for AVX instructions and virtualization technology (VT-x / AMD-V).

RAM

- Minimum: 32 GB RAM.
- Desirable: 64 GB and above, depending on the database size and application complexity.

Storage (SSD/HDD)

- Fast disk subsystem with high IOPS (input/output operations per second) throughput. NVMe or fast SSD is recommended.
- Disk volume should correspond to the size of databases plus a reserve for future expansion and storage of temporary files (log files, transaction logs, temporary sorting and index files).
- Use hardware RAID level 10 configurations or software analogs to increase fault tolerance and write/read speed.

3.3. Installation Process

The Digital Q.DataBase DBMS is delivered in 2 packages:

- DBMS server part (name qdatabase..) - contains the DBMS server and interactive SQL command lines (qsql, qclient);
- DBMS client part (name qdatabaseclient..) - contains interactive SQL command lines (qsql, qclient) and the Database Migration Wizard.

Packages are recommended to be installed on different machines to avoid resource shortages when starting the Database Migration Wizard.

Before installation, it is necessary to check that ports 5445, 1433, 1521 are not occupied on the server where the installation is performed.

The check is performed using the command (for all Linux OS distributions):

```
sudo netstat -tulpn
```

If any of the specified ports are busy, errors will occur during DBMS installation.

If it is impossible to free these ports – after a failed installation, it is necessary to change the used DBMS ports in the postgresql.conf file (details in section "2.4 File Structure") and restart all DBMS services (details in section "2.5 Digital Q.DataBase

Services")

3.3.1. Installation on Ubuntu OS

Steps for installing (removing) packages:

1. Connect the standard package repository of the current OS.
2. Download the required version of the DBMS Digital Q.DataBase deb-package.
3. Install deb-packages from the corresponding files (it is recommended to install the client package on a separate machine to distribute resources between the DBMS and the Database Migration Wizard):

```
sudo apt install ./qdatabase64xid-17-17.4.25112401-ubuntu22_x86_64.deb;  
sudo apt install ./qdatabaseclient64xid-17-17.4.25112401-ubuntu22_x86_64.deb;
```

4. If necessary to remove the server and client part packages, use the corresponding commands:

```
sudo apt remove qdatabase64xid-17;  
sudo apt remove qdatabaseclient64xid-17;
```

3.4. DBMS Installation Result

After successful installation, the Digital Q.DataBase server starts and becomes available on ports:

- 5445 (PostgreSQL dialect, PGWire protocol),
- 1433 (MS SQL dialect, TDS protocol),
- 1521 (Oracle Database dialect, PGWire protocol).

Attention! These ports will not be accessible if the connection to the DBMS occurs from outside! For remote (non-local) interaction of any application with the DBMS, it is required to add allowed IPs to the list in the pg_hba.conf file. Read more about this file further in the next subsection "File Structure".

All executable modules are located in the directory:

/opt/qdatabase-17

Databases are located in the directory:

`/var/lib/qdatabase-17/data`

DBMS operation logs are located in the directory:

`/var/log/qdatabase-17`

The client distribution creates shortcuts (Database Migration Wizard, documentation, and interactive SQL in the "Development" section) for OS versions with GUI. Executable files and documentation are located in the directory:

`/opt/qdb`

4 services start in the background: qdatabase-17, qdbproxy, qdbparse, qdbdgrid. By default, the following databases are created:

- qdb (for PostgreSQL dialect) - superuser "qdbadmin" with password "qdbadmin";
- master (for MS SQL dialect) - superuser "qdbadmin" with password "qdbadmin";
- qdbora (for Oracle Database dialect) - superuser "SYSDBA" with password "SYSDBA".

3.5. File Structure

3.5.1. Executable Files

All executable modules are located in the directory:

`/opt/qdatabase-17`

This directory contains files that simplify working with the PostgreSQL dialect:

1. **postgres**: Main database server process. It is responsible for processing queries and managing the database.
2. **pg_ctl**: Utility for managing the database server. Allows starting, stopping, and restarting the server.
3. **initdb**: Utility for initializing a new database. Creates a database cluster and configures it.
4. **createdb**: Utility for creating a new database in the cluster.
5. **createuser**: Utility for creating a new user (role).
6. **dropdb**: Utility for deleting a database from the cluster.

7. **dropuser**: Utility for deleting a user (role).
8. **pg_dump**: Utility for creating a backup of a database.
9. **pg_restore**: Utility for restoring a database from a backup created with pg_dump.
10. **pg_basebackup**: Utility for creating a full backup of a cluster.

3.5.2. Database Directory

All databases are located in the directory:

`/var/lib/qdatabase-17/data`

This directory contains configuration files. Among them, there are two important files:

1. `postgresql.conf`
2. `pg_hba.conf`

Let's consider their purpose.

File `postgresql.conf`

This is the main configuration file of the DB emulating PostgreSQL, containing global settings that affect the operation of the entire database system. It defines key performance parameters, memory, logging, replication, and other aspects of the DB emulating PostgreSQL.

File Structure

The file is organized into sections, each covering a specific group of parameters:

- General Settings
- File System Parameters
- Logging
- Performance and Memory
- Replication and Synchronization
- Security and Access Control

Parameters

The most significant parameters and their purposes are listed below:

General Settings

1. **listen_addresses:** Addresses on which the DB emulating PostgreSQL accepts connections. Default value is localhost. For external connections, specify " **or a specific IP address.** **listen_addresses = "**
2. **port:** Port for interaction with the DBMS in PostgreSQL dialect. Standard value is 5445. **port = 5445**
3. **qdatabasepg_tds.port:** Port for interaction with the DBMS in MS SQL dialect. Standard value is 1433. **qdatabasepg_tds.port = 1433**
4. **oport:** Port for interaction with the DBMS in Oracle Database dialect. Standard value is 1521. **oport=1521**
5. **smtp_host_port:** Host and port for interaction of the DBMS with an SMTP server (Oracle Database dialect, UTL_MAIL package). Standard value is 127.0.0.1:2525. **smtp_host_port='127.0.0.1:2525'**
6. **max_connections:** Maximum number of simultaneous connections to the database. Setting a value greater than 100 is recommended only for heavy loads. **max_connections = 100**

Performance and Memory Parameters

- **shared_buffers:** Size of the shared buffer area for storing data blocks. The more RAM available, the larger it can be set. **shared_buffers = 1GB**
- **effective_cache_size:** Estimated size of the disk cache available to the operating system. Set to about half of the physical RAM. **effective_cache_size = 4GB**
- **work_mem:** Amount of memory allocated for sorts and temporary tables. Increasing is recommended for complex queries. **work_mem = 64MB**
- **maintenance_work_mem:** Volume of memory reserved for administrative tasks like VACUUM and CREATE INDEX. **maintenance_work_mem = 512MB**

Logging

- **log_destination:** Where to write the error log (stderr, csvlog, syslog, etc.). **log_destination = 'csvlog'**
- **logging_collector:** Enables log collection into files. **logging_collector = on**
- **log_filename:** Template for log file names. **log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'**

Replication and Recovery

- **wal_level:** WAL (Write-Ahead Log) write level. Used for recovery and replication.
wal_level = replica
- **archive_mode:** Transaction log archiving mode for hot standby. archive_mode = on
- **archive_command:** Command executed to archive WAL files. archive_command = 'cp %p /path/to/wal_archive/%f'

Applying Changes

Changed parameters take effect after restarting the main DBMS service:

```
sudo systemctl restart qdatabase-17
```

Or using the SQL command:

```
SELECT pg_reload_conf();
```

Some parameters require a full service restart, which is indicated by a comment next to the parameter in the configuration file.

File pg_hba.conf

This main configuration file pg_hba.conf is used in the DBMS to define client authentication rules. These rules specify which users are allowed to connect to the database, from where they can connect, and which authentication method to use. File format:

TYPE	DATABASE	USER	ADDRESS	METHOD
local	all	qdbadmin		trust
local	all	SYSDBA		trust
local	all	all		md5
host	all	qdbadmin	127.0.0.1/32	trust
host	all	SYSDBA	127.0.0.1/32	trust
host	all	all	127.0.0.1/32	md5

Each line in the file corresponds to one authentication rule.

Field Descriptions:

- **Type** — defines the connection type:
- **local**: local connections via Unix sockets.
- **host**: TCP/IP connections with or without encryption.
- **hostssl**: requires SSL encryption for connection.
- **hostnossl**: prohibits SSL encryption.
- **Database** — the database to which the rule applies. You can specify specific database names, database groups (denoted by an asterisk *), or the word all for all databases.
- **User** — the user to whom this rule applies. Similarly to the previous field, individual user names, user groups (with asterisks), or all for all users are allowed.
- **Address** — the computer address or IP address range allowed access. Can be specified in CIDR format (e.g., 192.168.1.0/24), IPv6 address, or a single value for a specific IP address.
- **Method** — authentication method:
- **trust**: complete trust without password verification.
- **md5**: MD5 hash password verification.
- **password**: clear text password verification.
- **scram-sha-256**: modern authentication method with secure SHA-256 algorithm. Note: this method is not available when working via TDS protocol.
- **ident**: user identification via the OS.
- **ldap**: authentication via LDAP server.
- **cert**: certificate authentication.
- **Options** — optional parameters, such as session lifetime or LDAP server information.

Examples of entries in the pg_hba.conf file:

1. Allow local user qdbadmin to connect to all databases without a password

```
local all qdbadmin trust
```

2. Any users from IP 192.168.1.* can connect to the mydb database using md5

authentication

```
host mydb all 192.168.1.0/24 md5
```

3. Only user admin can connect to the datasecure database with an encrypted channel and scram-sha-256 method

```
host mydb all 192  
hostssl datasecure admin 172.16.0.0/16 scram-sha-256
```

4. All users can connect to any databases without encryption, using a standard password

```
host mydb all 192  
host all all 0.0.0.0/0 password
```

Rule Application Order:

Rules are applied sequentially from top to bottom. The first matching rule for the current connection is considered applied. Therefore, it is important to place stricter restrictions above less strict ones.

Changing Settings:

After modifying the pg_hba.conf file, it is necessary to reload the DB configuration emulating PostgreSQL with the command:

```
sudo systemctl reload qdatabase-17
```

or manually via SQL command:

```
SELECT pg_reload_conf();
```

3.5.3. Client Part Directory

The client part is by default located in the directory: /opt/qdb

The following files are located in this directory:

1. Executable files:

- mssql_migrator (launch DB migrator for MS SQL);
- oracle_migrator (launch DB migrator for Oracle);
- pg_migrator (launch DB migrator for Postgres);
- db_migrator_ctl (launch migrator for multiple schemas)
- db_migrator_gui (launch graphical interface of DB migrator);
- qsqlc.sh (launch terminal client qsql);
- pg_dump (utility for migrator work from PostgreSQL).

2. Configuration file templates for the Database Migration Wizard:

- cs_template.yaml (custom migration configuration template);
- mssql_template.yaml (configuration template for connecting to MS SQL Server and performing migration to Digital Q.DataBase);
- oracle_template.yaml (configuration template for connecting to Oracle Database and performing migration to Digital Q.DataBase);
- postgres_template.yaml (configuration template for connecting to PostgreSQL and performing migration to Digital Q.DataBase);
- qdb_ms_template.yaml (configuration template for connecting to Digital Q.DataBase when migrating from MS SQL Server);
- qdb_ora_template.yaml (configuration template for connecting to Digital Q.DataBase when migrating from Oracle Database);
- qdb_pg_template.yaml (configuration template for connecting to Digital Q.DataBase when migrating from PostgreSQL);

3.6. Digital Q.DataBase Services

3.6.1. Service qdatabase-17

qdatabase-17 is the main DBMS service, ensuring the operability of all its functional capabilities. This service is the core of the DBMS.

The DBMS core is the central component that ensures the execution of main data processing and management functions. It is responsible for processing user queries, ensuring data integrity, optimizing performance, and supporting concurrent work of multiple clients simultaneously.

3.6.2. Service qdbparse

qdbparse (interpreter) – a service for processing queries in the Oracle Database dialect, which the DBMS core cannot execute without additional transformations.

The interpreter contains the Oracle Database grammar, through which all queries are passed. If there are syntactic errors in the query, the interpreter will provide the user with information about it. If after syntax analysis errors related to the DBMS core occur (e.g., missing table), the interpreter will issue an error from the DBMS core.

Additionally, the interpreter processes procedures and functions of the Oracle Database dialect, ensuring their operation in the DBMS core.

Each time the service starts, a log file is created in the `/var/log/qdatabase-17/` directory with the start time indicated in the filename.

3.6.3. Service qdbproxy

The qdbproxy proxy server is an intermediate node between your device and the final server, accepting network requests and forwarding them to the target system. The main purpose of the proxy is to provide communication between the client and the server in such a way that the client thinks it is communicating directly with the original server.

The qdbproxy proxy server is implemented based on the PGWire protocol and serves to ensure smooth and secure interaction of client applications with the PostgreSQL core with the query processing system.

Functions of qdbproxy:

- **Remote Access Support:** users can connect to the database regardless of their location, which significantly facilitates migration and data management.
- **Increased Flexibility:** qdbproxy is capable of processing queries, changing their form, or filtering content before further redirection to the main database.
- **Improved Security:** since interaction with the main database passes through the proxy, the risk of unauthorized access from outside is reduced.
- **Simplified Architecture:** clients continue to use familiar drivers and utilities of the PostgreSQL dialect, retaining the full set of functional capabilities. Features of qdbproxy:
- **Connection Port:** the proxy constantly waits for incoming connections on port

1521 (can be changed in the postgresql.conf file).

- **Authorization:** access to qdbproxy is protected by a standard authentication procedure based on database user logins and passwords.
- **Query Processing:** after receiving a query from a client, qdbproxy directs it to the interpreter, which makes a decision about further processing. Sometimes the query is not sent to the database at all and is immediately returned to the client (e.g., service packets). Each time the service starts, a log file is created in the /var/log/qdatabase-17/ directory with the start time indicated in the filename.

3.6.4. Service qdbdgrid

qdbdgrid is a service of the built-in in-memory component DGrid, designed to organize data storage in RAM inside the DBMS process. The component provides mechanisms for prompt addition, retrieval, and verification of records, using memory as the primary storage. This solution is intended for use as a high-performance cache or buffer for working data inside the database, which allows speeding up the processing of frequently used information and reducing the load on the main table structure. Read more about working with the component later in a separate section.

4. Installation and Configuration of patroni

Patroni is a tool for managing PostgreSQL clusters, providing high availability (HA). It allows automating failover between the primary and standby database nodes.

1. The patroni service installation is performed after the Q.DataBase DBMS is installed, and the ETCD service is installed and started.

Stop the Q.DataBase service:

```
systemctl stop qdatabase-17
```

2. Go to the directory `/opt/patroni`

```
cd /opt/patroni
```

3. Start the installation with the command

```
sudo ./patroni_install.sh
```

4. Edit the file `/etc/patroni.yml` adding the necessary parameters to it.

Example of `patroni.yml` file

```
scope: dtrack-clust
name: dtrack-n1
namespace: /service/
restapi:
  listen: 0.0.0.0:8008
  connect_address: dtrack-n1:8008
etcd:
  hosts: dtrack-ha:2379,dtrack-n1:2379,dtrack-n2:2379
bootstrap:
method: initdb
dcs:
  ttl: 30
  loop_wait: 10
  retry_timeout: 10
  maximum_lag_on_failover: 1048576
  master_start_timeout: 300
  synchronous_mode: false
```

```
synchronous_mode_strict: false
synchronous_node_count: 1
postgresql:
  use_pg_rewind: true
  use_slots: true
  parameters:
    max_connections: 5000
    superuser_reserved_connections: 5
    password_encryption: md5
    max_locks_per_transaction: 10000
    max_prepared_transactions: 0
    huge_pages: try
    shared_buffers: 8000MB
    work_mem: 16MB
    maintenance_work_mem: 256MB
    effective_cache_size: 16GB
    checkpoint_timeout: 60min
    checkpoint_completion_target: 0.9
    min_wal_size: 80MB
    max_wal_size: 4GB
    wal_buffers: 32MB
    default_statistics_target: 100
    seq_page_cost: 1
    random_page_cost: 3.0
    effective_io_concurrency: 3
    synchronous_commit: on
    autovacuum: on
    autovacuum_max_workers: 5
    autovacuum_vacuum_scale_factor: 0.2
    autovacuum_analyze_scale_factor: 0.1
    autovacuum_vacuum_cost_limit: -1
    autovacuum_vacuum_cost_delay: 2
    autovacuum_naptime: 1m
    max_files_per_process: 1000
    archive_mode: on
    archive_timeout: 1800s
    archive_command: cd .
    wal_level: replica
    wal_keep_size: 2GB
    max_wal_senders: 10
    max_replication_slots: 10
    hot_standby: on
    wal_log_hints: on
    wal_compression: on
    shared_preload_libraries: pg_stat_statements
    pg_stat_statements.max: 10000
    pg_stat_statements.track: all
```

```
pg_stat_statements.track_utility: on
pg_stat_statements.save: true
auto_explain.log_min_duration: 10s
auto_explain.log_analyze: true
auto_explain.log_buffers: true
auto_explain.log_timing: false
auto_explain.log_triggers: true
auto_explain.log_verbose: true
auto_explain.log_nested_statements: true
track_io_timing: on
log_lock_waits: on
log_temp_files: -1
track_activities: on
track_counts: on
track_functions: all
log_checkpoints: on
logging_collector: on
log_truncate_on_rotation: on
log_rotation_age: 1d
log_rotation_size: 0
log_line_prefix: '%t [%p-%l] %r %q%u@%d '
log_filename: 'postgresql-%a.log'
log_directory: /data/postgres/pgdata/log/
hot_standby_feedback: on
max_standby_streaming_delay: 30s
wal_receiver_status_interval: 10s
idle_in_transaction_session_timeout: 5min
jit: off

initdb: # List options to be passed on to initdb
- encoding: UTF8
- locale: en_US.UTF-8
- data-checksums

pg_hba: # Add following lines to pg_hba.conf after running 'initdb'
- host replication postgres ::1/128 md5
- host replication postgres 127.0.0.1/8 md5
- host replication postgres 10.20.110.59/32 trust # dtrack-n1
- host replication postgres 10.20.110.60/32 trust # dtrack-n2
- host all all 0.0.0.0/0 md5

postgresql:
listen: 0.0.0.0:5432
connect_address: dtrack-n1:5432
use_unix_socket: true
data_dir: /data/postgres/pgdata/
bin_dir: /opt/qdatabase-17/bin/
config_dir: /data/postgres/pgdata/
pgpass: /data/postgres/.pgpass_patroni
authentication:
```

```
replication:
  username: postgres
  password: postgres
superuser:
  username: postgres
  password: postgres
parameters:
  unix_socket_directories: /tmp/
remove_data_directory_on_rewind_failure: false
remove_data_directory_on_diverged_timelines: false
create_replica_methods:
  - basebackup
basebackup:
  max-rate: '100M'
  checkpoint: 'fast'
watchdog:
  mode: off # Allowed values: off, automatic, required
  device: /dev/watchdog
  safety_margin: 5
tags:
nofailover: false
noloadbalance: false
clonefrom: false
nosync: false
```

5. Edit the connection settings file `/var/lib/qdatabase-17/data/pg_hba.conf`, while it is necessary to ensure connectivity to other nodes.

Example of `pg_hba.conf` file

```
# TYPE  DATABASE      USER      ADDRESS      METHOD
# "local" is for Unix domain socket connections only
local   all           all                md5
# IPv4 local connections:
host    all           all          127.0.0.1/32  md5
host    all           all          0.0.0.0/0     md5
# IPv6 local connections:
host    all           all          ::1/128       md5
host    all           all          ::0/0         md5
# Allow replication connections from localhost, by a user with the
# replication privilege.
local   replication  all                md5
host    replication  all          127.0.0.1/32  md5
host    replication  all          0.0.0.0/0     md5
host    replication  all          ::1/128       md5
host    replication  all          ::0/0         md5
```

6. Start the patroni service on the first node.

```
systemctl start patroni
```

7. Check that the service is running.

```
systemctl status patroni
```

8. Check that the node is running.

```
patronictl -c /etc/patroni.yml list
```

9. Start the **patroni** service on the second node.

It should be noted that the second node will be synchronized via pg_basebackup, so the directory specified in `data_dir` must be empty.

```
systemctl start patroni
```

10. Check that the service is running.

```
systemctl status patroni
```

11. Check that the node is running.

```
patronictl -c /etc/patroni.yml list
```

12. To switch between replication servers, use the following command:

```
patronictl -c /etc/patroni.yml switchover
```

5. Determining Q.DataBase Version

In Q.DataBase, there are two functions for displaying the current product version.

Function version

The `version()` function completely replicates the behavior of the original PostgreSQL function and contains detailed information about the PostgreSQL version, including the version number and additional build details.

Example usage:

```
qdb=# SELECT version();
version
-----
Digital Q.DataBase 17.4 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 12.3.0-
1ubuntu1~22.04.2) 12.3.0, 64-bit
(1 row)
```

Function qdb_version

The `qdb_version()` function outputs the product name and version, as well as the build number, license information, and additional build details.

Example usage:

```
qdb=# SELECT qdb_version();
qdb_version
-----
Digital Q.DataBase 17.4 on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 12.3.0-
1ubuntu1~22.04.2) 12.3.0, 64-bit, Module name: POSTGREDB, License expiration date:
2026-06-05 00:00:00, Available cores: 8
(1 row)
```

6. License

Q.DataBase has license protection that complies with the company's product standards. To obtain a license, a request is created to the CTL service or through an account manager, which sends an xml file indicating the license expiration date, product (module), unique signature, and other parameters.

6.1. Warnings about License Expiration

The licensing block includes a warning system for license expiration:

1. License expiration 30 days before expiry.

When detecting license expiration in 30 days, the licensing block issues a warning once a day in the DBMS log (by default — `/var/log/qdatabase-17/pgdata-..log`) with the following message:

```
License will expire in %ld days.  
Contact your supplier for updates.
```

where %ld is the number of days until license expiration

2. License expiration is overdue.

When a license is detected as expired, the DBMS functionality continues to operate normally, but the licensing block will issue a warning every 5 minutes in the DBMS log (by default — `/var/log/qdatabase-17/pgdata-..log`) with the following message:

```
License has expired. Contact your supplier for updates.
```

6.2. Updating the License File

To update the license file, it is necessary to:

1. Copy the new license file with replacement to the directory `/var/lib/qdatabase-17/data`.
2. The license will be updated automatically — the server re-reads the file every 5 minutes.

Server restart is not required.

7. SQL Dialect Features

The DBMS provides support for several SQL dialects through corresponding communication protocols: 1. **TDS protocol**: provides access to the MS SQL Server dialect. 2. **PGWire protocol**: provides the use of the PostgreSQL and Oracle Database dialects, as well as partial support for MS SQL Server dialects.

7.1. Using the Oracle Database Dialect in Digital Q.DataBase

To connect to a DB emulating the structure and data types of an Oracle Database, the **PGWire** protocol is used.

PGWire ensures a transparent connection between client and server, abstracting from specific network protocols and platforms.

To access the Oracle Database dialect, port **1521** is used by default.

To connect to Digital Q.DataBase via this protocol, the qclient client application for Linux and Windows OS is available.

To run it, you can use various command-line formation methods, providing flexibility in connection configuration. Consider the available options:

1) Full command specifying schema and port:

```
qclient qdb://SYSDBA:SYSDBA@localhost:1521/qdbora
```

where:

*qdb:// — connection type; *SYSDBA:SYSDBA — username and password;
*localhost:1521 — host and port (Oracle Database dialect); *qdbora — database name.

2) Interactive mode:

```
qclient
```

As a result, the interactive mode will start, allowing you to manually enter all necessary parameters (login, password, host, port, etc.).

7.2. Using the MS SQL Server Dialect in Digital Q.DataBase

To connect to a DB emulating the structure and data types of MS SQL Server, the **TDS** protocol is used.

TDS allows clients (applications) to exchange data with a database server in a structured format

To access the MS SQL Server dialect, port **1433** is used by default.

To connect via **TDS**, a user can use any client application that allows interaction with the DBMS via this protocol. One of the standard client applications for working with the MS SQL Server dialect is SQL Server Management Studio (SSMS) (the list of available functionality for these applications is described in the section "10. Working with the DBMS through external applications").

Connection Methods

Any client tool using a compatible driver with support for TDS protocol version 7.1 and higher is capable of connecting to Digital Q.DataBase. Exceptions are cases where the Bulk Copy API is used, which is not currently supported.

Supported drivers for connection are:

1. SQL Server Native Client 11.0.
2. Open Database Connectivity (ODBC)
3. OLEDB Provider/MSOLEDBSQL
4. JDBC version 8.2.2 and higher (MS SQL-jdbc-8.2.2+)
5. Microsoft SqlClient Data Provider for SQL Server
6. .NET Data Provider for SQL Server
7. OLEDB Driver/SQLOLEDB

The SQL dialect has broad capabilities and includes a wide variety of operators, functions, and constructs for solving various tasks. Here are the main categories of capabilities:

1. Mathematical Functions

- Performing basic arithmetic operations: addition (+), subtraction (-), multiplication (*), division (/), remainder (%).

- Support for advanced mathematical functions: square roots (SQRT()), exponentials (EXP()), logarithms (LOG(), LOG10()), trigonometric functions (SIN(), COS(), TAN()).
- Number rounding: round up (CEILING()), round down (FLOOR()), regular rounding (ROUND()).

2. Strings and Text Processing

- String concatenation with the + operator or CONCAT() function.
- Changing letter case: UPPER(), LOWER().
- Searching and replacing string fragments: SUBSTRING(), REPLACE(), LEFT(), RIGHT(), LEN().
- Removing extra characters: TRIM(), LTRIM(), RTRIM().

3. Date and Time

- Date and time operations: comparison, selecting part of a date (YEAR(), MONTH(), DAY()), intervals (DATEADD(), DATEDIFF()).
- Date-time representation formats: CONVERT(), CAST().

4. Conditional Constructs and Logic

- Standard comparison operators: equality (=), inequality (<>), less than (<), greater than (>).
- Boolean logic: AND, OR, NOT.
- Support for conditional selection operator: CASE WHEN ... THEN ... ELSE ... END.

5. Aggregate Functions

- Summation (SUM()), average value (AVG()), minimum/maximum value (MIN(), MAX()), record count (COUNT()).
- Extended aggregate functions: window functions (OVER(), PARTITION BY, ranking functions ROW_NUMBER(), RANK(), DENSE_RANK()).

6. Transactional Capabilities

- Transaction support: start (BEGIN TRANSACTION), commit (COMMIT), rollback (ROLLBACK).
- Isolation level locks: serializable (SERIALIZABLE), read committed (READ COMMITTED), repeatable read (REPEATABLE READ).

7. Storing Business Logic

- Creating and using stored procedures (CREATE PROCEDURE).
- Implementing triggers (CREATE TRIGGER), executed when data changes.
- Using built-in functions and query optimization methods: indexing, plan caching, support for clustered and non-clustered indexes.

8. Security and Audit

- Authentication and authorization mechanisms: support for user roles, privilege levels, data encryption.
- User activity audit: audit log, tracking changes in data and schemas.

9. Advanced Capabilities

- Working with spatial data: geometry, geography (GEOMETRY, GEOGRAPHY).
- Integration with programming languages: CLR integration (.NET Framework).
- High-performance mechanisms for parallel query execution and distributed data storage (AlwaysOn Availability Groups, Distributed Query Processing).

Features of MS SQL Server Dialect Implementation in Digital Q.DataBase

- **Partial Support for SQL Server Functions:** Some SQL Server functions, such as CREATE AGGREGATE, CREATE FULLTEXT CATALOG, and working with cryptographic keys, are not supported in Digital Q.DataBase.
- **Unavailable Data Types:** Specific SQL Server data types, such as ROWVERSION, GEOMETRY, and GEOGRAPHY, are not supported.
- **Lack of Full-Featured Indexes:** Spatial and full-text indexes, widely used in SQL Server, are not supported in Digital Q.DataBase.
- **Working with Permissions and Security:** Role and account management is significantly limited in Digital Q.DataBase compared to SQL Server.
- **Incomplete Implementation of Stored Procedures:** SQL Server stored procedures, functions, and triggers do not fully replicate their behavior in Digital Q.DataBase.
- **Index and Indexed View Limitations:** Creating indexed views and performing high-level query optimization, as in SQL Server, is not possible.
- **Insufficient Support for Global Temporary Tables:** The concept of global

temporary tables is not implemented in Digital Q.DataBase.

- **Other Limitations:** Built-in SQL Server functions such as SYSTEM_USER, CURRENT_TIMESTAMP, and temporary table structure are not supported.

Other Functional Capabilities

Support for the following functionality is also planned in the future:

1. *SQL Server Replication* — a technology designed for automatic distribution and updating of data between multiple SQL Server instances or other database systems. It allows organizing consistent data distribution, increasing reliability, availability, and scalability of the information system.
2. *DBCC (Database Console Commands)* — a group of instructions intended for diagnostics, integrity checking, and performing administrative tasks related to database maintenance. They allow checking data integrity, defragmenting indexes, collecting statistical information, and much more.
3. *SQL Profiler* — a tool designed for monitoring, logging, and analyzing actions occurring on the database server. It allows tracking events, collecting statistics on query execution, monitoring user activity, and identifying performance bottlenecks.

7.3. Using the PostgreSQL Dialect in Digital Q.DataBase

To connect to a DB emulating the structure and data types of PostgreSQL, the **PGWire** protocol is used.

PGWire (also known as PostgreSQL Wire Protocol) is a network protocol used for interaction between client applications and the PostgreSQL database server. It ensures data exchange between client and server in a structured format.

To access the **PostgreSQL** dialect, port **5445** must be used.

To connect to Digital Q.DataBase via this protocol, the `qsql` client application is used. For more details, see section "4. Working with Digital Q.DataBase using the `qsql` client application". Also, part of the functionality of the Oracle Database and MS SQL Server dialects is available in this dialect (for more details, see section "10. Built-in Functions of MS SQL Server and Oracle Database Dialects in PostgreSQL").

Built-in Functions of the Oracle Database Dialect

1. ADD_MONTHS
2. LAST_DAY
3. LENGTH
4. LNNVL
5. MONTHS_BETWEEN
6. NEXT_DAY
7. NLSSORT
8. NVL
9. NVL2
10. ROUND
11. SESSIONTIMEZONE
12. SUBSTR
13. SYS_EXTRACT_UTC
14. SYSDATE FROM DUAL
15. TO_CHAR
16. TO_DATE
17. TO_MULTI_BYTE
18. TO_SINGLE_BYTE
19. TRUNC

Also, the DUAL table syntax is available in the PostgreSQL dialect.

DUAL is a special table used for executing queries that do not require specifying a real table. It contains one row and one column, allowing various operations such as calculations, conversions, and other functions to be performed without accessing real data.

Usage Example:

```
SELECT SYSDATE FROM DUAL;
```

8. Data Types in Digital Q.DataBase

Digital Q.DataBase supports the following data types:

1. Numeric

Type Name	Size (bytes)	Values	Description	Note
TINYINT	1	0..255	Integer type	analog of type TINYINT in MS
SQLSMALLINT	2	-32768 .. +32767	Integer type	
INTEGER	4	-2147483648 .. +2147483647	Integer type	
OINTEGER	17	$-10^{38+1} .. 10^{38-1}$	Integer type	analog of type INTEGER in Oracle Database corresponds to decimal(38,0)
BIGINT	8	-9223372036854775808 .. 9223372036854775807	Integer type	
DECIMAL[(p [, s])]	5/9/13/17	$-10^{38+1} .. 10^{38-1}$	Fixed-point number	
NUMERIC[(p [, s])]	5/9/13/17	$-10^{38+1} .. 10^{38-1}$	Fixed-point number - up to 131072 digits before decimal, up to 16383 digits after decimal.	
NUMBER[(p [, s])]	$1 + ((p/2) + 1) / \text{up to } 21$	$-10^{38+1} .. 10^{38-1}$	Fixed-point number - up to 131072 digits before decimal, up to 16383 digits after decimal.	analog of type NUMBER[(p [, s])] in Oracle Database
REAL	4	$-3.40\text{E}+38 .. 3.40\text{E}+38$, precision within 6 decimal digits	Floating-point number	

Type Name	Size (bytes)	Values	Description	Note
DOUBLE PRECISION	8	–1.79E+308 to 1.79E+308, precision within 15 decimal digits	Floating-point number	
FLOAT[(n)]	(4/8)\(1..22)	–1.79E+308 to 1.79E+308, precision within 15 decimal digits	Fixed-point number, n sets number of mantissa bits, max=126.	analog of type FLOAT[(n)] in Oracle Database
BINARY_FLOAT	4	–3.40E+38 .. 3.40E+38	Floating-point number	analog of type BINARY_FLOAT in Oracle Database
BINARY_DOUBLE	8	–1.79E+308 to 1.79E+308	Floating-point number	analog of type BINARY_DOUBLE in Oracle Database
MONEY	8	-922337203685477 58.08 .. +922337203685477 58.07	Fixed-point number (for currency)	
SMALLMONEY	4	-214 748.3648..214 748.3647	Fixed-point number (for currency)	analog of type SMALLMONEY in MS SQL
BIT	1	0..1	Binary values	analog of type BIT in MS SQL
SMALLSERIAL	2	1 .. 32767	Auto-increment integer type	
SERIAL	4	1 .. 2147483647	Auto-increment integer type	
BIGSERIAL	8	1 .. 9223372036854775 807	Auto-increment integer type	
IDENTITY	4	1 .. 2147483647	Auto-increment integer type	analog of type IDENTITY in MS SQL

Type Name	Size (bytes)	Values	Description	Note
BIGINT IDENTITY	8	1 .. 9223372036854775 807	Auto-increment integer type	analog of type BIGINT IDENTITY in MS SQL
AUTOINCREMENT	4 / 8	1 .. 9223372036854775 807	Auto-increment integer type	analog of type AUTOINCREMENT in Oracle
SIMPLE_FLOAT	4	-3.40E+38 .. 3.40E+38	Floating-point number	analog of type BINARY_FLOAT in Oracle Database
SIMPLE_DOUBLE	8	-1.79E+308 to 1.79E+308	Floating-point number	analog of type BINARY_DOUBLE in Oracle Database
BINARY_INTEGER	4	-2147483648 .. +2147483647	Integer type	analog of type BINARY_INTEGER in Oracle Database (PL/SQL)
PLS_INTEGER	4	-2147483648 .. +2147483647	Integer type	analog of type PLS_INTEGER in Oracle Database (PL/SQL)
SIMPLE_INTEGER	4	-2147483648 .. +2147483647	Integer type	analog of type SIMPLE_INTEGER in Oracle Database (PL/SQL)

II. String

Type Name	Size (bytes)	Encoding	Description	Note
CHAR(n),BPCHAR(n)	n * <character size>, up to 4000	ASCII/UTF-8	Fixed-size text of n characters in ASCII/UTF-8 encoding.	
OCHAR(n)	n * <character size>, up to 4000	ASCII/UTF-8	Fixed-size text of n characters in ASCII/UTF-8 encoding.	analog of type CHAR(n [BYTE \ CHAR]) in Oracle Database

Type Name	Size (bytes)	Encoding	Description	Note
BPCHAR	up to 1 GB	ASCII/UTF-8	Unlimited variable-length string with spaces removed	
VARCHAR(n)	n * <character size> + 1, up to 4000	ASCII/UTF-8	Variable length up to n characters.	
OVARCHAR(n)	n * <character size> + 1, up to 4000	ASCII/UTF-8	Text, size specified in bytes or characters, can additionally specify measurement unit "by default" maximum data length. Special behavior with empty string - it is not equal to NULL.	analog of type VARCHAR(n [BYTE \ CHAR]) in Oracle Database
TEXT	1..2147483647	database code page	Variable-length text, for large texts.	
LONG	up to 4 GB	UTF-8/UTF-16	For storing large text objects	analog of type CLOB in Oracle Database (PL/SQL)
RAW	up to 4 GB	UTF-8/UTF-16	For storing large text objects	analog of type CLOB in Oracle Database (PL/SQL)
VARCHAR2(size [BYTE \ CHAR])	n + 1, up to 32767	UTF-8/UTF-16	Text, size specified in bytes or characters, can additionally specify measurement unit "by default" maximum data length. Special behavior with empty string - it is not equal to NULL.	analog of type VARCHAR2(n [BYTE \ CHAR]) in Oracle Database

Type Name	Size (bytes)	Encoding	Description	Note
NCHAR(n)	2 * n (up to 8000)	UTF-16	Fixed-size text with Unicode support.	analog of type NCHAR(n) in MS SQL
ONCHAR(n)	2 * n (up to 8000)	UTF-16	Fixed-size text with Unicode support. Special behavior with empty string - it is not equal to NULL.	analog of type NCHAR(n) in Oracle Database
NVARCHAR(n\max)	2 * n + 2(4000\2^31-1)	UTF-16	Variable-length text up to n characters in UTF-16. Parameter "max" means up to 2 GB. Default length 1 character.	analog of type NVARCHAR(n\max) in MS SQL
NVARCHAR2(n)	n * (2\3) (up to 4000 \ 32767)	UTF-8/UTF-16	Variable-length text, size specified in characters, size up to 32767 bytes in EXTENDED mode and 4000 bytes - STANDARD	analog of type NVARCHAR2(n) in Oracle Database
NTEXT	1..2147483647	UTF-16	Variable-length text for large texts in UTF-16.	analog of type NTEXT in MS SQL
JSON	up to 1 GB	UTF-8	Storing JSON data	
OJSON	up to 1 GB	UTF-8	Storing JSON data	analog of type JSON in Oracle Database
JSONB	up to 1 GB	UTF-8	Binary storage of JSON for fast search	
XML	up to 1 GB Postgres / 2 GB MS SQL	database code page	For storing XML data	

Type Name	Size (bytes)	Encoding	Description	Note
XMLTYPE	up to 4 GB \ 128 TB	UTF-8/UTF-16	For storing texts with XML support	analog of type XMLType in Oracle Database
CLOB	up to 4 GB	UTF-8/UTF-16	For storing large text objects	analog of type CLOB in Oracle Database
NCLOB	up to 4 GB	UTF-16	For storing large texts with Unicode support	analog of type NCLOB in Oracle Database (PL/SQL)
UUID	16	Binary	For storing unique identifiers in PostgreSQL. Analog in Oracle Database - RAW(16)	
UNIQUEIDENTIFIER	16	Binary	For storing unique identifiers in MS SQL	analog of type UNIQUEIDENTIFIER in MS SQL
CHARACTER(n)	n * <character size>, up to 4000	ASCII/UTF-8	Fixed-size text of n characters in ASCII/UTF-8 encoding.	analog of type CHAR(n [BYTE \ CHAR]) in Oracle Database (PL/SQL)
CHARACTER_VARYING	n * <character size>, up to 4000	ASCII/UTF-8	Fixed-size text of n characters in ASCII/UTF-8 encoding.	analog of type CHAR(n [BYTE \ CHAR]) in Oracle Database (PL/SQL)
CHAR_VARYING	n * <character size>, up to 4000	ASCII/UTF-8	Fixed-size text of n characters in ASCII/UTF-8 encoding.	analog of type CHAR(n [BYTE \ CHAR]) in Oracle Database (PL/SQL)
NATIONAL_CHARACTER	n * (2\3) (up to 4000 \ 32767)	UTF-8/UTF-16	Variable-length text, size specified in characters, size up to 32767 bytes in EXTENDED mode and 4000 bytes - STANDARD	analog of type NVARCHAR2(n) in Oracle Database (PL/SQL)

Type Name	Size (bytes)	Encoding	Description	Note
NATIONAL_CHAR	n * (2\3) (up to 4000 \ 32767)	UTF-8/UTF-16	Variable-length text, size specified in characters, size up to 32767 bytes in EXTENDED mode and 4000 bytes - STANDARD	analog of type NVARCHAR2(n) in Oracle Database (PL/SQL)
CHARACTER_LARGE_OBJECT	up to 4 GB	UTF-8/UTF-16	For storing large text objects	analog of type CLOB in Oracle Database (PL/SQL)
CHAR_LARGE_OBJECT	up to 4 GB	UTF-8/UTF-16	For storing large text objects	analog of type CLOB in Oracle Database (PL/SQL)
NATIONAL_CHARACTER_LARGE_OBJECT	up to 4 GB	UTF-8/UTF-16	For storing large text objects	analog of type CLOB in Oracle Database (PL/SQL)
NCHAR_LARGE_OBJECT	up to 4 GB	UTF-8/UTF-16	For storing large text objects	analog of type CLOB in Oracle Database (PL/SQL)

III. Date/Time

Type Name	Size (bytes)	Description	Note
DATE	8	Stores date (year, month, day)	
ODATE	8	Stores date (year, month, day) and time hours:minutes:seconds.	analog of type DATE in Oracle Database
TIME	8	Stores only time (hours, minutes, seconds), with microsecond precision	
TIME WITH TIME ZONE	12	Stores time with time zone (does not save the time zone itself)	
TIMESTAMP	8	Stores date and time with microsecond precision	

Type Name	Size (bytes)	Description	Note
OTIMESTAMP	8	Stores date and time with microsecond precision	analog of type TIMESTAMP in Oracle Database
TIMESTAMP_TZ/TIMESTAMP WITH TIME ZONE	12	Stores date, time, and time zone.	
OTIMESTAMP_TZ	12	Stores date, time, and time zone.	analog of type TIMESTAMP WITH TIME ZONE in Oracle Database
OTIMESTAMP_LTZ[(fractional_seconds_precision)]	12	Stores date, time, and local time zone.	analog of type TIMESTAMP WITH LOCAL TIME ZONE in Oracle Database
SMALLDATETIME	4	Stores date and time with minute precision, range from 1900 to 2079	
DATETIME	8	Stores date and time with 3.33 millisecond precision	
DATETIME2	8	Stores date and time with 100 nanosecond precision	
SYSTIMESTAMP	8	Stores current timestamp with time zone	analog of type SYSTIMESTAMP in Oracle Database
CURRENT_TIMESTAMP	8	Stores current date and time with time zone	
CURRENT_DATE	4	Stores current date without time	
CURRENT_TIME	8	Stores current time without date	
INTERVAL	16	Stores time interval, supports years, months, days, hours, minutes, seconds	

Type Name	Size (bytes)	Description	Note
OINTERVAL_Y2M	5	Stores time intervals expressed in years and months.	analog of type INTERVAL YEAR TO MONTH in Oracle Database
OINTERVAL_D2S	11	Stores time intervals expressed in days, hours, minutes, seconds, and fractions of seconds.	analog of type INTERVAL DAY TO SECOND in Oracle Database
TIME_UNCONSTRAINED	8	Stores only time (hours, minutes, seconds), with microsecond precision	analog of type TIME in Oracle Database (PL/SQL)
TIME_TZ_UNCONSTRAINED	8	Stores only time (hours, minutes, seconds), with microsecond precision	analog of type TIME in Oracle Database (PL/SQL)
TIMESTAMP_UNCONSTRAINED	12	Stores date, time, and time zone.	analog of type TIMESTAMP WITH TIME ZONE in Oracle Database (PL/SQL)
TIMESTAMP_TZ_UNCONSTRAINED	12	Stores date, time, and time zone.	analog of type TIMESTAMP WITH TIME ZONE in Oracle Database(PL/SQL)
YMINTERVAL_UNCONSTRAINED	5	Stores time intervals expressed in years and months.	analog of type INTERVAL YEAR TO MONTH in Oracle Database (PL/SQL)
DSINTERVAL_UNCONSTRAINED	11	Stores time intervals expressed in days, hours, minutes, seconds, and fractions of seconds.	analog of type INTERVAL DAY TO SECOND in Oracle Database (PL/SQL) TIMESTAMP_LTZ_UNCONSTRAINED

IV. Other

Type Name	Size (bytes)	Description	Note
BOOLEAN	1	Boolean type	

Type Name	Size (bytes)	Description	Note
ROWID	10	Unique row identifier in a table	analog of type ROWID in Oracle Database
UROWID	up to 4000	Extended version of ROWID, can reference external rows.	analog of type UROWID in Oracle Database
HIERARCHYID	-	Storing data for hierarchical structures, e.g., trees	analog of type HIERARCHYID in MS SQL
GEOGRAPHY	up to 2 GB	Stores geographic data in point, line, polygon format.	analog of type GEOGRAPHY in MS SQL
SDO_GEOMETRY	-	Stores geometric data for working with spatial data	analog of type SDO_GEOMETRY in Oracle Database
ARRAY	-	Stores arrays of any supported type	
VARRAY	-	Stores arrays of any supported type	analog of type VARRAY in Oracle Database
OTABLE	-	Stores tables	analog of type TABLE in Oracle Database
SET	-	Stores a set of values	analog of type SET in Oracle Database
BLOB	128 TB	Binary data	analog of type BLOB in Oracle Database
LONG_RAW	128 TB	Binary data	analog of type BLOB in Oracle Database
BINARY_LARGE_OBJECT	128 TB	Binary data	analog of type BLOB in Oracle Database
REFCURSOR		Pointer to a cursor	
OREFCURSOR		Pointer to a cursor	analog of type REF CURSOR in Oracle Database (PL/SQL)
RECORD		Structure	

Type Name	Size (bytes)	Description	Note
ORECORD		Structure	analog of type RECORD in Oracle Database (PL/SQL)
OTABLE		Table	analog of type TABLE in Oracle Database (PL/SQL)

9. Working with Digital Q.DataBase using the qsql client application

9.1. Creating a Database

To work with the PostgreSQL dialect in Digital Q.DataBase, the **qsql** client (terminal) application is used. This application allows full-fledged work with the DBMS within the PostgreSQL dialect. If PostgreSQL is not present on the server, psql is also installed along with qsql, which is an alternative to qsql.

The process of creating a database in qsql consists of several simple steps.

1. Changing the active Linux OS user

By default, when installing Digital Q.DataBase, a system user named qdbadmin is created. To log into its session, execute the command:

```
sudo su - qdbadmin
```

2. Creating a new database

After logging into the session as the qdbadmin user, create a new database using the qcreatedb utility. For example, let's name our database mydb:

```
qcreatedb mydb;
```

If you need to assign another user as the database owner, specify it explicitly:

```
qcreatedb -* new_owner mydb;
```

where new_owner is the username who owns the database.

3. Entering the Digital Q.DataBase shell

To start working with the created database, enter the Digital Q.DataBase terminal client:

```
qsql -U qdbadmin -d qdb -h localhost -p 5445 -c "select 1;"
```

Now you are in the interactive **qsql** environment, where you can execute SQL queries.

4. Basic commands for working with a database

- Show list of all databases:

```
\l
```

- List tables in the current database:

```
\d
```

- Exit from the Digital Q.DataBase shell:

```
\q
```

5. Finishing work After finishing work with the database, leave the qdbadmin user session: exit

9.2. Creating Database Objects

1. Displaying information about tables:

- **\dt**: Displays a list of all tables in the current schema.

```
\dt
```

- **\d+ tablename**: Displays detailed information about the table structure, including fields, indexes, constraints, and comments.

```
\d+ users
```

- **\z tablename**: Displays access rights to the specified table.

```
\z users
```

2. Commands for working with triggers Displaying information about triggers:

- **\dT**: Shows a list of all triggers in the current database.

```
\dT
```

- **\dT+ triggername:** Shows detailed information about a specific trigger.

```
\dT+ track_user_changes_trigger
```

3. Commands for working with stored procedures and functions Displaying information about functions and procedures:

- **\df:** Shows a list of all functions and procedures in the current database. **\df**
- **\df+ function_or_procedure_name:** Shows detailed information about a specific procedure or function. **\df+ register_new_user**

4. Commands for working with aggregate functions Displaying information about aggregate functions:

- **\da:** Shows a list of all aggregate functions in the current database. **\da**
- **\da+ aggregate_function_name:** Shows detailed information about a specific aggregate function. **\da+ avg_age**

Working with objects in the terminal

Steps for working with objects via terminal:

1. Starting the Digital Q.DataBase terminal (qsql):

```
qsql-h localhost -d your_database -U your_user
```

Where:

- **your_database** — your database name.
- **your_user** — username.);

2. Viewing created objects: To view a list of tables:

```
\d
```

To view table structure:

```
\d+ users
```

To view lists of indexes and views:

```
\dI+  
\dV+
```

3. Finishing work:

To exit qsql, enter:

```
\q
```

10. Description of Database Objects

10.1. Creating Tables

Q.DataBase supports standard SQL types: int, smallint, real, double precision, char(N), varchar(N), date, time, timestamp and interval, as well as other universal types and a set of geometric types. Furthermore, Q.DataBase can be extended by creating custom data types. Consequently, type names are not keywords in this notation, except when required to implement special SQL standard constructs.

Q.DataBase supports various SQL dialects, including Oracle Database (PL/SQL), MS SQL Server (T-SQL), and standard PostgreSQL.

To create a table, you must specify its name, list all column names and their types:

Oracle Database Dialect:

```
CREATE TABLE BINARY_DATA_0 (  
    id NUMBER PRIMARY KEY,  
    file_name VARCHAR2(100),  
    upload_date DATE DEFAULT SYSDATE,  
    description VARCHAR2(200)  
);
```

MS SQL Server Dialect:

```
CREATE TABLE employees (  
    employee_id INT IDENTITY(1,1) PRIMARY KEY,  
    first_name NVARCHAR(50),  
    last_name NVARCHAR(50),  
    hire_date DATETIME,  
    salary DECIMAL(10,2)  
);
```

PostgreSQL Dialect:

```
CREATE TABLE employees (  
    employee_id SERIAL PRIMARY KEY,  
    first_name VARCHAR(50),  
    last_name VARCHAR(50),  
    hire_date DATE,  
    salary NUMERIC(10,2)  
);
```

Differences in syntax and dialect features:

- **Auto-increment:** Oracle - GENERATED ALWAYS AS IDENTITY, MS SQL - IDENTITY(x,y), PostgreSQL - SERIAL
- **Data types:** Oracle - VARCHAR2, NUMBER; MS SQL - NVARCHAR, DATETIME; PostgreSQL - VARCHAR, NUMERIC
- **Primary keys:** All dialects support PRIMARY KEY
- **Unicode:** MS SQL uses NVARCHAR, other DBMSs use standard VARCHAR with UTF-8

If a table is no longer needed or needs to be recreated differently, it can be deleted using the following command:

```
DROP TABLE table_name;
```

10.2. Creating Indexes

Indexes are a means of increasing database performance. Using an index, the database server can find and retrieve required rows much faster than without it. However, indexes impose an additional load on the DBMS as a whole, so they should be applied thoughtfully. Updating a table with indexes takes longer than updating a table without indexes (since indexes also need to be updated). Therefore, indexes should only be created for columns that will be frequently searched.

Syntax for creating an index for a table:

Examples of creating indexes in different dialects:

MS SQL Server Dialect:

```
CREATE NONCLUSTERED INDEX idx_employees_last_name ON employees(last_name);  
CREATE NONCLUSTERED INDEX idx_employees_hire_date ON employees(hire_date);
```

PostgreSQL Dialect:

```
* allows duplicate values:  
CREATE INDEX index_name  
ON table_name (column1, column2, ...);  
* does not allow duplicate values:  
CREATE UNIQUE INDEX index_name  
ON table_name (column1, column2, ...);
```

Indexing methods: PostgreSQL allows explicit selection of indexing method (BTREE, HASH, GIN, etc.) To delete an index, use the DROP INDEX command. Indexes can be added and removed at any time.

Once an index is created, no additional actions are required. The system will update it automatically when table data changes, and will use it in queries where, in its opinion, it is more efficient than scanning the entire table. Indexes can also be useful when executing UPDATE and DELETE commands with search conditions. Furthermore, they can be used in join searches. That is, an index defined for a column participating in a join condition can significantly speed up queries with JOIN.

Creating an index for a large table can take a long time. By default, Q.DataBase allows reading (SELECT statements) of the table in parallel with index creation, but write operations (INSERT, UPDATE, and DELETE) are blocked until index building is complete. For a production environment, this limitation is often unacceptable.

Q.DataBase supports several index types: B-tree, hash, GiST, SP-GiST, GIN, and BRIN. Different index types use different algorithms, oriented towards specific types of queries. By default, the CREATE INDEX command creates B-tree indexes, effective in most cases. To choose a different type, write the index type name after the USING keyword. For example, to create a hash index:

```
CREATE INDEX index_name ON table_name USING HASH (column1);
```

10.3. Foreign Keys

Foreign keys are used for relationships between tables. A foreign key is set for a

column from a dependent, child table (referencing table), and points to one of the columns from the main table (referenced table). Typically, a foreign key points to the primary key of the related main table.

General syntax for setting a foreign key at the column level:

```
REFERENCES main_table (column_main_table)
[ON DELETE {CASCADE|RESTRICT}]
[ON UPDATE {CASCADE|RESTRICT}]
```

To establish a relationship between tables, after the REFERENCES keyword, the name of the related table is specified, followed in parentheses by the name of the column from that table that the foreign key will point to. After the REFERENCES expression, ON DELETE and ON UPDATE expressions may follow, which specify behavior when data is deleted or updated. General syntax for setting a foreign key at the table level:

```
FOREIGN KEY (column1, column2, column3, ...)
REFERENCES main_table (column1_main_table, column2_main_table, column3_main_table,
...)
[ON DELETE {CASCADE|RESTRICT}]
[ON UPDATE {CASCADE|RESTRICT}]
```

10.4. Uniqueness Constraints

Uniqueness constraints guarantee that data in a specific column or group of columns is unique among all rows in the table.

A feature of implementing uniqueness constraints in Q.DataBase is the ability to define uniqueness not only at the level of a column value but also for an expression on a column value.

General syntax for creating a table with a uniqueness constraint:

```
CREATE TABLE table_name
(
    ...
    [ CONSTRAINT constraint_name ]
    UNIQUE [ NULLS [ NOT ] DISTINCT ] (expression)
    UNIQUE [ NULLS [ NOT ] DISTINCT ] (expression , ... )
    [INCLUDE (column_name [, ...])] (table_constraint)
);
```

Example of creating a table with a uniqueness constraint for the absolute value of a column:

```
CREATE TABLE reference (
    cost INTEGER,
    CONSTRAINT reference_cost_constraint UNIQUE(abs(cost)) DEFERRABLE INITIALLY
    DEFERRED
);
```

In this example, the uniqueness constraint is set for the expression `abs(cost)`, where `cost` is the column name of the reference table.

The `DEFERRABLE` (`NOT DEFERRABLE`) clause determines whether a constraint can be deferred. A non-deferrable constraint (`NOT DEFERRABLE`) is checked immediately after each command. Checking of deferrable constraints (`DEFERRABLE`) can be postponed. By default, `NOT DEFERRABLE` is implied.

For deferrable constraints, the `INITIALLY IMMEDIATE` and `INITIALLY DEFERRED` clauses are used to define when the constraint should be checked. A constraint with the characteristic `INITIALLY IMMEDIATE` (the default) is checked after each statement. Conversely, a constraint with `INITIALLY DEFERRED` is checked only at the end of the transaction. When performing a bulk insert of rows into a table for a column that had a uniqueness constraint with the `INITIALLY IMMEDIATE` clause, uniqueness validation will be performed for each inserted row. If a uniqueness constraint with the `INITIALLY DEFERRED` clause was specified for a table column, uniqueness validation will be performed after inserting all rows.

Example of adding a uniqueness constraint after creating a table:

```
CREATE TABLE reference (  
    cost INTEGER  
);  
ALTER TABLE reference ADD CONSTRAINT reference_cost_constraint UNIQUE(abs(cost))  
DEFERRABLE INITIALLY DEFERRED;  
Example of adding a uniqueness constraint for multiple table columns:  
CREATE TABLE reference (  
    cost INTEGER,  
    add_cost INTEGER  
);  
ALTER TABLE reference ADD CONSTRAINT reference_cost_constraint UNIQUE(add_cost,  
abs(cost));
```

In this example, the uniqueness constraint is set for the expression `abs(cost)` and the `add_cost` column, where `cost` is the column name of the reference table.

10.5. Additional Database Objects

Creating Views (VIEW):

Oracle Database Dialect:

```
CREATE VIEW TEST_VIEW_1 AS  
SELECT id, file_name, upload_date, description  
FROM (SELECT 1 as id, 'test_file_1' as file_name, SYSDATE as upload_date, 'Test  
description 1' as description FROM DUAL);
```

PostgreSQL Dialect:

```
CREATE VIEW employee_department_view AS  
SELECT e.employee_id, e.first_name, e.last_name, d.department_name  
FROM employees e JOIN departments d ON e.department_id = d.department_id;
```

Materialized Views:

```
CREATE MATERIALIZED VIEW department_salary_summary AS  
SELECT d.department_id, d.department_name, SUM(e.salary) as total_salary  
FROM employees e  
JOIN departments d ON e.department_id = d.department_id  
GROUP BY d.department_id, d.department_name;
```

Partitioned Tables:

PostgreSQL Dialect:

```
CREATE TABLE employees (  
    employee_id SERIAL PRIMARY KEY,  
    first_name TEXT NOT NULL,  
    hire_date TIMESTAMP WITHOUT TIME ZONE NOT NULL  
)  
PARTITION BY RANGE(hire_date);
```

CREATE TABLE employees_p2018 PARTITION OF employees FOR VALUES FROM ('2018-01-01') TO ('2019-01-01'); Spatial data (with PostGIS): CREATE EXTENSION IF NOT EXISTS postgis;

CREATE TABLE spatial_data (id SERIAL PRIMARY KEY, geom geometry(Point, 4326));

Hierarchical Collection Types:

PostgreSQL Dialect:

```
CREATE TYPE EmployeeInfo AS (  
    employee_id INTEGER,  
    full_name TEXT  
);
```

CREATE TABLE departments (dept_id SERIAL PRIMARY KEY, department_name TEXT, employees EmployeeInfo[]);

11. Data Modification

11.1. Adding Data

After creating a table, you need to add data to it. You must specify column names and values for insertion. The command syntax:

```
INSERT INTO table_name (column1, column2, column3...)
VALUES (value1, value2, value3, ...);
```

If values for all table columns are being added, you do not need to specify column names in the SQL query. However, the order of values must correspond to the order of columns in the table. Here the INSERT INTO syntax will be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

Data values are listed in the order of columns in the table and separated by commas. One command can insert multiple rows at once:

```
INSERT INTO table_name (column1, column2, column3...) VALUES
(value1, value2, value3, ...),
(value4, value5, value6, ...),
(value7, value8, value9, ...);
```

It is also possible to insert the result of a query (which may contain no rows, one, or several):

```
INSERT INTO table_name (column1, column2, column3...)
SELECT value1, value2, value3 FROM table_name2
WHERE condition;
```

Examples in different dialects:

Oracle Database Dialect:

```
INSERT INTO employees (employee_id, first_name, last_name)
VALUES (1, 'John', 'Doe');
```

MS SQL Server Dialect:

```
INSERT INTO employees (employee_id, first_name, last_name)
VALUES (1, 'John', 'Doe');
```

PostgreSQL Dialect:

```
INSERT INTO employees (employee_id, first_name, last_name)
VALUES (1, 'John', 'Doe')
RETURNING employee_id;
```

11.2. Modifying Data

Modifying data already stored in the database is called updating. You can update all rows of a table, a subset of all rows, or only selected rows. Each column can be modified independently of others.

To change data in existing rows, the UPDATE command is used. It requires the following information:

1. Table name and column to be modified;
2. New column value;
3. Criteria for selecting rows to modify.

The command syntax will be as follows:

```
UPDATE table_name
SET column1 = value1, column2 = value2,...
WHERE condition;
```

You must specify the conditions the desired row must meet. Only if the table has a primary key (regardless of whether you declared it or not), can you uniquely address individual rows by defining a condition on the primary key.

The WHERE clause specifies which records need to be updated. If the WHERE clause is omitted, all records in the table will be updated.

Data update examples:

Oracle Database Dialect:

```
sql
UPDATE employees
SET salary = salary * 1.1
WHERE employee_id = 1;
_MS SQL Server Dialect:_
```

```
[%unbreakable]UPDATE e
SET salary = e.salary * 1.1
FROM employees e
JOIN departments d ON e.department_id = d.id
WHERE d.name = 'Sales';
```

11.3. Selecting Data

To retrieve data from a table, you need to execute a query. For this, the SQL SELECT operator is intended. It consists of several parts: selection (listing the columns to be retrieved), a list of tables (listing the tables from which data will be retrieved), and an optional condition (defining constraints). The returned data is stored in a result table, called a result set.

The command syntax is as follows:

```
SELECT column1, column2,...
FROM table_name;
```

Here column1, column2, ... are the names of the table fields from which data is selected. If you need to select all fields available in the table, use the following syntax: SELECT * FROM table_name;

Here * is a shorthand for "all columns".

The query can be supplemented with a condition by adding a WHERE clause, limiting the set of returned rows. The WHERE clause specifies a logical expression (truth test), which serves as a row filter: only those rows for which this expression is true end up in the result. The syntax in this case will be as follows:

```
SELECT column1, column2,...  
FROM table_name  
WHERE conditions;
```

If necessary, duplicate rows can be removed from the query result:

```
SELECT DISTINCT column1, column2,...  
FROM table_name;
```

11.4. Deleting Data

Rows can also be deleted from a table using the DELETE operator. Data can only be deleted in whole rows, just as they can be added. To delete selected rows, you must formulate appropriate conditions for them. But if the table has a primary key, you can uniquely identify a specific row using it. You can also delete groups of rows matching a condition, or all rows of the table at once.

To delete rows, the DELETE operator is used; its syntax is very similar to that of the UPDATE operator:

```
DELETE FROM table_name WHERE condition;  
At the same time, be careful with statements like  
DELETE FROM table_name;
```

Without a condition, DELETE will delete all rows of the given table, completely clearing it. The system will not ask you to confirm the operation.

Data deletion examples:

Oracle Database Dialect:

```
DELETE FROM employees WHERE employee_id = 1;
```

MS SQL Server Dialect:

```
DELETE FROM employees WHERE employee_id = 1;
```

PostgreSQL Dialect:


```
DELETE FROM employees WHERE employee_id = 1;
```

11.5. Additional Data Modification Operations

Data Merging (MERGE/UPSERT):

PostgreSQL Dialect:

```
INSERT INTO target (id, value)
VALUES (1, 'New Value')
ON CONFLICT (id) DO UPDATE
SET value = EXCLUDED.value;
```

Bulk Operations:

MS SQL Server Dialect:

```
INSERT INTO target_table
SELECT *
FROM source_table;
```

PostgreSQL Dialect:

```
BEGIN TRANSACTION;
SET LOCAL synchronous_commit TO OFF;
TRUNCATE TABLE target_table;
INSERT INTO target_table SELECT * FROM source_table;
COMMIT;
```

Exception Handling:

MS SQL Server Dialect (TRY-CATCH):

```
BEGIN TRY
    BEGIN TRAN;
    DELETE FROM users WHERE id = 1;
    COMMIT TRAN;
END TRY
BEGIN CATCH
    ROLLBACK TRAN;
    THROW;
END CATCH;
```

PostgreSQL Dialect:

```
BEGIN
    PERFORM 1 / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'Cannot divide by zero!';
END $$ LANGUAGE plpgsql;
```

12. Grouping and Sorting Data

12.1. WHERE Clause

The WHERE clause is used to filter records. It is used to extract only those records that satisfy a specified condition. The syntax is: WHERE condition;

where condition is any expression yielding a boolean result.

After processing the FROM clause, each row of the resulting virtual table is checked against the constraint condition. If the condition result is true, this row remains in the output table; otherwise (if the result is false or NULL), it is discarded. The constraint condition typically involves at least one column from the table output by FROM. Used not only in SELECT statements, but also in UPDATE, DELETE, etc.

```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition;
```

12.1.1. Logical Operators

The set of logical operators includes the usual ones: boolean AND boolean → boolean
boolean OR boolean → boolean NOT boolean → boolean

SQL works with a three-state logical system: TRUE, FALSE, and NULL ("unknown" state).

The WHERE clause can be combined with the AND, OR, and NOT operators. The AND and OR operators are used to filter records based on multiple conditions:

- The AND operator displays a record if all conditions separated by AND are true;
- The OR operator displays a record if any of the conditions separated by OR is true;
- The NOT operator displays a record if the condition(s) is NOT true.

The AND and OR operators are commutative, meaning that changing the order of operands does not change the result.

```
SELECT column1, column2,...  
FROM table_name  
WHERE condition1 AND condition2 AND condition3 ...;
```

```
SELECT column1, column2,...  
FROM table_name  
WHERE condition1 OR condition2 OR condition3 ...;
```

```
SELECT column1, column2,...  
FROM table_name  
WHERE NOT condition;
```

12.2. GROUP BY Clause

Rows of the generated input table that have passed the WHERE filter can be grouped using the GROUP BY clause.

```
SELECT column_name(s)  
FROM table_name  
WHERE condition  
GROUP BY column_name(s);
```

The GROUP BY clause groups table rows, combining them into one group when values in all listed columns match. The order in which columns are specified does not matter. As a result, sets of rows with identical values are transformed into separate rows representing all rows of the group. This can be useful for eliminating redundancy in output data and/or for computing aggregate functions applied to these groups.

The GROUP BY operator is often used with aggregate functions COUNT, MAX, MIN, SUM, AVG to group a result set by one or more columns.

In a grouped table, columns not included in the GROUP BY list can only be used in aggregate expressions. Example of such an aggregate expression:

```
SELECT x, sum(y) FROM test1 GROUP BY x;
```

Expected result:

text x | sum ---+----- a | 4 b | 5 c | 2 (3 rows)

Here sum is an aggregate function calculating a single value for the entire group. Grouping without aggregate expressions yields a set of distinct column values. The same result can be achieved using the DISTINCT clause.

In the SQL standard, GROUP BY can only group by columns of the source table, but the Q.DataBase extension allows using columns from the selection list in GROUP BY. Grouping by expressions, not just column names, is also possible.

12.2.1. Aggregate Functions

Like most other relational database servers, Q.DataBase supports aggregate functions. An aggregate function computes a single value by processing multiple rows. They are also very useful in combination with the GROUP BY clause.

The MIN function returns the smallest value of the selected column.

The MAX function returns the largest value of the selected column.

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

The COUNT function returns the number of rows matching a given criterion.

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

The AVG function returns the average value of a numeric column.

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

The SUM function returns the total sum of a numeric column.

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

12.3. ORDER BY Clause

After the query produces a result table (after processing the selection list), it can be sorted. If sorting is not specified, rows are returned in an undefined order. The actual row order in this case will depend on the join and scan plan, as well as the order of data on disk, so it cannot be relied upon. A defined order of output rows is guaranteed only if a sorting step is explicitly specified. The sort order is determined by the ORDER BY clause:

```
SELECT column1, column2,...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

The ORDER BY keyword sorts records in ascending order by default. To sort records in descending order, use the DESC keyword. Sort expressions can be any expressions allowed in the query's selection list. For example:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

When multiple expressions are specified, subsequent values allow sorting rows where all previous values matched. To determine the placement of NULL values, the NULLS FIRST and NULLS LAST specifications can be used, placing NULL values before or after non-NULL values, respectively. By default, NULL values are considered greater than any other, meaning NULLS FIRST for DESC order and NULLS LAST otherwise are implied.

12.4. Additional Data Processing Capabilities

Recursive Queries (WITH RECURSIVE):

PostgreSQL Dialect:

```
WITH RECURSIVE EmpHierarchy AS (  
    SELECT EmployeeID, ManagerID, EmployeeName  
    FROM Employees  
    WHERE ManagerID IS NULL  
    UNION ALL  
    SELECT e.EmployeeID, e.ManagerID, e.EmployeeName  
    FROM Employees e  
    INNER JOIN EmpHierarchy eh ON e.ManagerID = eh.EmployeeID  
)  
SELECT * FROM EmpHierarchy;
```

Limiting the Number of Rows:

Oracle Database Dialect:

```
SELECT first_name, last_name  
FROM employees  
ORDER BY salary DESC  
FETCH FIRST 10 ROWS ONLY;
```

MS SQL Server Dialect:

```
[%unbreakable]SELECT TOP(10) FirstName, LastName  
FROM Employees  
ORDER BY Salary DESC;
```

PostgreSQL Dialect:

```
[%unbreakable]SELECT first_name, last_name  
FROM employees  
ORDER BY salary DESC  
LIMIT 10;
```

Joins (JOIN):

Inner join:

```
SELECT e.first_name, d.department_name  
FROM employees e  
INNER JOIN departments d ON e.department_id = d.department_id;
```

Left outer join:

```
SELECT e.first_name, d.department_name  
FROM employees e  
LEFT OUTER JOIN departments d ON e.department_id = d.department_id;
```

Full outer join:

```
SELECT e.first_name, d.department_name  
FROM employees e  
FULL OUTER JOIN departments d ON e.department_id = d.department_id;
```


13. Transactions

Transactions are a fundamental concept in all DBMSs. The essence of a transaction is that it combines a sequence of actions into a single "all-or-nothing" operation. Intermediate states within the sequence are invisible to other transactions, and if something prevents the transaction from completing successfully, none of the results of these actions are saved to the database.

A transaction is atomic from the perspective of other transactions. It either executes and commits completely, or does not commit at all.

A transactional database guarantees that all changes are written to permanent storage before the transaction is considered complete. Changes made by an open transaction are invisible to other transactions until it is completed, and then they become visible all at once.

In Q.DataBase, a transaction is defined by a set of SQL commands surrounded by BEGIN and COMMIT commands. A banking transaction might look like this:

```
BEGIN;  
UPDATE table_name SET column1 = column2 - 100.000 WHERE conditions;  
...  
COMMIT;
```

If during transaction execution it is decided not to commit its changes, you can execute the ROLLBACK command instead of COMMIT, and all changes will be undone.

Transaction examples in different dialects:

Oracle Database Dialect:

```
BEGIN;  
    UPDATE table_name SET column1 = column2 - 100.000 WHERE conditions;  
    ...  
COMMIT;  
_MS SQL Server Dialect:_
```

```
BEGIN TRANSACTION;  
    UPDATE employees SET salary = salary * 1.1 WHERE department_id = 10;  
COMMIT TRANSACTION;  
_PostgreSQL Dialect:_
```

```
BEGIN;  
    UPDATE employees SET salary = salary * 1.1 WHERE department_id = 10;  
COMMIT;
```

13.1. Transaction Isolation

The SQL standard defines four levels of transaction isolation. The most stringent of these, serializable, is defined in one paragraph stating that when executed concurrently, several serializable transactions must be guaranteed to produce the same result as if they were run sequentially in some order. The other three levels are defined by describing special phenomena that may occur during interaction of concurrent transactions but are not allowed at a certain level. As noted in the standard, it follows from the definition of the serializable level that none of these phenomena are possible at this level.

The standard describes the following special conditions, not allowed for various isolation levels.

"Dirty" Read

A transaction reads data written by a concurrent uncommitted transaction. Non-repeatable Read: A transaction re-reads the same data as before and finds that it has been modified by another transaction (which committed after the first read).

Phantom Read

A transaction re-executes a query returning a set of rows for some condition and finds that the set of rows satisfying the condition has changed due to a transaction that committed in the meantime.

Serialization Anomaly

The result of successful commit of a group of transactions turns out to be inconsistent under all possible scenarios of executing these transactions sequentially.

Transaction Isolation Levels

Isolation Level	"Dirty" Read	Non-repeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in Q.DataBase	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in Q.DataBase	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

In Q.DataBase, you can request any of the four transaction isolation levels; however, internally only three distinct levels are implemented, meaning the Read Uncommitted mode in Q.DataBase acts like Read Committed.

To select the desired transaction isolation level, use the SET TRANSACTION command. Read Committed is the default transaction isolation level in Q.DataBase. In a transaction operating at this level, a SELECT query (without a FOR UPDATE/SHARE clause) sees only data that was committed before the query began; it never sees uncommitted data or changes made during query execution by concurrent transactions. SELECT sees a snapshot of the database at the moment the query begins execution. However, SELECT sees the results of changes made earlier in the same transaction, even if they are not yet committed. Two consecutive SELECT statements may see different data even within the same transaction if some other transactions commit changes after the first SELECT starts but before the second starts.

The UPDATE, DELETE, SELECT FOR UPDATE, and SELECT FOR SHARE commands behave similarly to SELECT when searching for target rows: they find only those target

14. SQL Query Usage Examples

14.1. Simple Queries

Let's consider basic examples of using simple SQL queries in three dialects: Oracle Database, MS SQL Server, PostgreSQL. Below are typical SELECT query examples illustrating the execution of basic operations on tables for each format.

Oracle Database

Let's create the necessary tables (employees and departments):

```
CREATE TABLE employees (  
    employee_id INT PRIMARY KEY,  
    first_name VARCHAR(50) NOT NULL,  
    last_name VARCHAR(50) NOT NULL,  
    department_id INT,  
    salary DECIMAL(10, 2)  
);
```

```
CREATE TABLE departments (  
    department_id INT PRIMARY KEY,  
    department_name VARCHAR(100) NOT NULL  
);
```

Next, let's move on to INSERT, UPDATE, DELETE operations:

```
-- Adding new records to the employees table.  
INSERT INTO employees (employee_id, first_name, last_name, department_id, salary)  
VALUES  
    (1, 'Ivan', 'Ivanov', 1, 50000.00);  
INSERT INTO employees (employee_id, first_name, last_name, department_id, salary)  
VALUES  
    (2, 'Anna', 'Petrova', 1, 60000.00);  
INSERT INTO employees (employee_id, first_name, last_name, department_id, salary)  
VALUES  
    (3, 'Boris', 'Semenov', 2, 45000.00);  
INSERT INTO employees (employee_id, first_name, last_name, department_id, salary)  
VALUES  
    (4, 'Elena', 'Krasnova', 2, 55000.00);  
INSERT INTO employees (employee_id, first_name, last_name, department_id, salary)  
VALUES  
    (5, 'Mikhail', 'Novikov', 3, 70000.00);
```

```
-- Adding new records to the Departments table.  
INSERT INTO departments (department_id, department_name) VALUES (1, 'IT Department');  
INSERT INTO departments (department_id, department_name) VALUES (2, 'Marketing');  
INSERT INTO departments (department_id, department_name) VALUES (3, 'Sales');  
INSERT INTO departments (department_id, department_name) VALUES (4, 'Finance');
```

```
-- Updating employee data  
UPDATE employees  
SET salary = 50000  
WHERE employee_id = 2;
```

```
-- Deleting an employee  
DELETE FROM employees  
WHERE employee_id = 2;
```

Next, let's move on to queries:

```
-- Selecting all records from the employees table  
SELECT employee_id, first_name, last_name  
FROM employees;
```

```
-- Grouping by departments and getting average salary
SELECT department_id, AVG(salary)
FROM employees
GROUP BY department_id;
```

```
-- Joining two tables (employees and departments)
SELECT e.employee_id, e.first_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

MS SQL Server

Let's create the necessary tables (employees and departments):

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    DepartmentID INT,      -- restoring DepartmentID field
    RegionID INT          -- adding RegionID field
);
```

```
CREATE TABLE Departments (
    DepartmentID INT PRIMARY KEY,
    DepartmentName NVARCHAR(100) NOT NULL
);
```

Next, let's move on to INSERT, UPDATE, DELETE operations:

```
INSERT INTO Departments (DepartmentID, DepartmentName)
VALUES
    (1, 'Marketing'),
    (2, 'IT'),
    (3, 'Accounting');
```

```
INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID, RegionID)
VALUES
    (1, 'Ivan', 'Ivanov', 1, 1),
    (2, 'Anna', 'Smirnova', 1, 1),
    (3, 'Boris', 'Semenov', 2, 2),
    (4, 'Elena', 'Petrovna', 2, 2),
    (5, 'Kirill', 'Vasiliev', 3, 3);
```

```
UPDATE Employees
SET RegionID = 5
WHERE EmployeeID = 2;
```

```
DELETE FROM Employees
WHERE EmployeeID = 2;
```

Next, let's move on to queries:

```
-- Selecting all employees
SELECT EmployeeID, FirstName, LastName
FROM Employees;
```

```
-- Aggregating by regions
SELECT RegionID, COUNT(*) AS TotalEmployees
FROM Employees
GROUP BY RegionID;
```

```
-- Joining two tables (Employees and Departments)
SELECT E.EmployeeID, E.FirstName, D.DepartmentName
FROM Employees E
INNER JOIN Departments D ON E.DepartmentID = D.DepartmentID;
```

PostgreSQL

Let's create the necessary tables (employees and departments):

```
CREATE TABLE Departments (  
    DepartmentID SERIAL PRIMARY KEY,  
    DepartmentName VARCHAR(100) NOT NULL  
);
```

```
CREATE TABLE Employees (  
    EmployeeID SERIAL PRIMARY KEY,  
    FirstName VARCHAR(50) NOT NULL,  
    LastName VARCHAR(50) NOT NULL,  
    DepartmentID INT REFERENCES Departments(DepartmentID),  
    Salary DECIMAL(10, 2)  
);
```

Next, let's move on to INSERT, UPDATE, DELETE operations:

```
INSERT INTO Employees (FirstName, LastName, DepartmentID, Salary)  
VALUES  
    ('Ivan', 'Ivanov', 1, 50000.00),  
    ('Anna', 'Smirnova', 1, 60000.00),  
    ('Boris', 'Semenov', 2, 45000.00),  
    ('Elena', 'Petrovna', 2, 55000.00),  
    ('Kirill', 'Vasiliev', 3, 70000.00);
```

```
INSERT INTO Departments (DepartmentName)  
VALUES  
    ('Marketing'),  
    ('Sales'),  
    ('Finance');
```

```
UPDATE employees  
SET salary = 50000  
WHERE EmployeeID = 2;
```

```
-- Deleting an employee  
DELETE FROM employees  
WHERE EmployeeID = 2;
```

Next, let's move on to queries:


```
-- Simple record selection
SELECT id, first_name, last_name
FROM employees;
```

```
-- Aggregating data by departments
SELECT dept_id, AVG(salary)
FROM employees
GROUP BY dept_id;
```

```
-- Inner join of tables
SELECT emp.id, emp.first_name, dep.dept_name
FROM employees emp
JOIN departments dep ON emp.dept_id = dep.id;
```

14.2. Complex Data Operations

Moving on to a deeper study of DBMS capabilities, let's consider a number of complex operations that will help solve a wide range of real tasks.

1) Oracle Database

Task:

Monitor that goods from the warehouse, brought to the store "for display," are returned back on time. How it works: a laptop is brought to the store, it is entered into the system, and a return date is automatically set for 2 months later. Consultants see the product description and when it needs to be returned. The system automatically checks — is it time to return the product to the warehouse.

```
REM Table of store goods arrived from the warehouse
CREATE TABLE product_descriptions (
    product_id NUMBER PRIMARY KEY, --product id
    product_name VARCHAR2(100), --product name
    description CLOB, --product description
    return_date DATE, --date to send back to warehouse
    created_date DATE DEFAULT SYSDATE --creation date of the accepted product card
);
REM Package with 2 procedures
CREATE OR REPLACE PACKAGE product_utils AS
    --Adding a new product to the system
    PROCEDURE add_product_description(
```

```

        p_id NUMBER, --product id
        p_name VARCHAR2, --product name
        p_text VARCHAR2 --product description
    );
    --Displaying product information by id
    PROCEDURE show_description(
        p_id NUMBER --product id
    );
END product_utils;
/
REM Package implementation
CREATE OR REPLACE PACKAGE BODY product_utils AS
    PROCEDURE add_product_description(
        p_id NUMBER,
        p_name VARCHAR2,
        p_text VARCHAR2
    ) IS
        v_clob CLOB; --Local variables
        v_current_date DATE;
    BEGIN
        DBMS_LOB.CREATETEMPORARY(v_clob, TRUE); --Create a temporary CLOB
        DBMS_LOB.WRITE(v_clob, LENGTH(p_text), 1, p_text); --Write text p_text into
v_clob
        SELECT SYSDATE INTO v_current_date FROM DUAL; --Write current date into
variable
        --Save everything to the table, adding 2 months to the send-to-warehouse date:
        INSERT INTO product_descriptions (product_id, product_name, description,
return_date)
        VALUES (p_id, p_name, v_clob, ADD_MONTHS(v_current_date, 2));
        DBMS_LOB.FREETEMPORARY(v_clob); --Free the temporary CLOB
    END;
    PROCEDURE show_description(
        p_id NUMBER
    ) IS
        v_clob CLOB; --Local variables
        v_buffer VARCHAR2(4000);
        v_amount_to_read NUMBER := 4000;
        v_product_name VARCHAR2(100);
        v_return_date DATE;
    BEGIN
        SELECT product_name INTO v_product_name FROM product_descriptions WHERE
product_id = p_id;
        SELECT description INTO v_clob FROM product_descriptions WHERE product_id =
p_id;
        DBMS_LOB.READ(v_clob, v_amount_to_read, 1, v_buffer); --Read first 4000
characters
        SELECT return_date INTO v_return_date FROM product_descriptions WHERE

```

```

product_id = p_id;
    --Output result
    DBMS_OUTPUT.PUT_LINE('Product name: ' || v_product_name);
    DBMS_OUTPUT.PUT_LINE('Product description: ' || v_buffer);
    DBMS_OUTPUT.PUT_LINE('Date to send back to warehouse: ' || v_return_date);
END;
END product_utils;
/
REM Out-of-package function for checking product return date expiration
CREATE OR REPLACE FUNCTION is_return_date_expired(
    p_product_id NUMBER
) RETURN VARCHAR2
IS
    v_return_date DATE;
    v_result VARCHAR2(100);
BEGIN
    SELECT return_date INTO v_return_date
    FROM product_descriptions
    WHERE product_id = p_product_id;
    IF v_return_date < SYSDATE THEN
        v_result := 'Return date to warehouse HAS EXPIRED';
    ELSE
        v_result := 'Return date to warehouse NOT expired';
    END IF;
    RETURN v_result;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 'Product with specified ID not found';
END is_return_date_expired;
/
SET SERVEROUTPUT ON
REM Test block
DECLARE
    v_product_id NUMBER := 1; -- product id
    v_product_name VARCHAR2(100) := 'ASUS TUF Gaming A15';
    v_description VARCHAR2(4000) := '1920x1080, IPS, AMD Ryzen 5 7535HS, cores: 6, RAM
16 GB, SSD 512 GB, GeForce RTX 2050 for laptops 4 GB, no OS';
BEGIN
    -- Clear table before test
    DELETE FROM product_descriptions WHERE product_id = v_product_id;
    -- Add product description
    product_utils.add_product_description(v_product_id, v_product_name,
v_description);
    -- View description
    product_utils.show_description(v_product_id);
    -- Check product return date using out-of-package function
    DBMS_OUTPUT.PUT_LINE('Return date check: ' ||

```

```
is_return_date_expired(v_product_id));  
    -- Clear table after test  
    DELETE FROM product_descriptions WHERE product_id = v_product_id;  
END;  
/  
REM Cleanup objects  
DROP PACKAGE product_utils;  
DROP FUNCTION is_return_date_expired;  
DROP TABLE product_descriptions;
```

Main elements:

- product_descriptions table - storing products with large descriptions (CLOB)
- product_utils package - contains procedures:
- add_product_description - creates a temporary CLOB, writes data into it, and saves to table
- show_description - reads CLOB and outputs product information
- is_return_date_expired function - checks if product return date has expired

Key technologies:

- working with temporary CLOBs (DBMS_LOB.CREATETEMPORARY);
- LOB read/write operations (DBMS_LOB.WRITE, DBMS_LOB.READ);
- managing large text data;
- package approach to code organization.

2) MS SQL Server

Task:

Automating sales reporting for computer equipment

Organize a system that automatically calculates revenue, shows the most profitable products, and calculates manager bonuses. This will allow quick decision-making on purchases and motivate employees.

```
-- Setting date format  
SET dateformat dmy  
go  
-- Creating table for sales accounting
```

```
IF OBJECT_ID('Sales', 'U') IS NOT NULL
    DROP TABLE Sales
go
CREATE TABLE Sales (
    SaleID INT IDENTITY(1,1) PRIMARY KEY,
    ProductName NVARCHAR(100) NOT NULL,
    SaleDate DATE DEFAULT GETDATE(),
    Quantity INT NOT NULL,
    UnitPrice DECIMAL(10,2) NOT NULL,
    TotalAmount AS (Quantity * UnitPrice), -- Computed field
    SaleMonth AS (FORMAT(SaleDate, 'yyyy-MM')) -- Computed field for analysis
)
go
-- Filling with test data
INSERT INTO Sales (ProductName, SaleDate, Quantity, UnitPrice) VALUES
    ('ASUS VivoBook Laptop', '15.01.2024', 2, 45000.00),
    ('Wireless mouse', '18.01.2024', 5, 1200.00),
    ('24" Monitor', '22.01.2024', 1, 18000.00),
    ('Mechanical keyboard', '15.02.2024', 3, 3500.00),
    ('Lenovo IdeaPad Laptop', '28.02.2024', 1, 52000.00),
    ('Mouse pad', '05.03.2024', 10, 500.00)
go
-- Procedure for sales analysis for a period
IF OBJECT_ID('GetSalesReport', 'P') IS NOT NULL
    DROP PROCEDURE GetSalesReport
go
CREATE PROCEDURE GetSalesReport
    @StartDate DATE,
    @EndDate DATE
AS
BEGIN
    -- Overall statistics for the period
    SELECT
        COUNT(*) as 'Number of sales',
        SUM(TotalAmount) as 'Total revenue',
        AVG(TotalAmount) as 'Average check',
        MAX(TotalAmount) as 'Maximum sale',
        MIN(TotalAmount) as 'Minimum sale'
    FROM Sales
    WHERE SaleDate BETWEEN @StartDate AND @EndDate
    -- Product breakdown
    SELECT
        ProductName as 'Product',
        SUM(Quantity) as 'Quantity',
        SUM(TotalAmount) as 'Amount',
        FORMAT(SUM(TotalAmount) * 100.0 / (SELECT SUM(TotalAmount) FROM Sales WHERE
        SaleDate BETWEEN @StartDate AND @EndDate), 'N2') + '%' as 'Share in revenue'
```

```
FROM Sales
WHERE SaleDate BETWEEN @StartDate AND @EndDate
GROUP BY ProductName
ORDER BY SUM(TotalAmount) DESC
-- Sales by month
SELECT
    SaleMonth as 'Month',
    COUNT(*) as 'Number of sales',
    SUM(TotalAmount) as 'Revenue'
FROM Sales
WHERE SaleDate BETWEEN @StartDate AND @EndDate
GROUP BY SaleMonth
ORDER BY SaleMonth
END
go
-- Procedure for calculating manager bonuses
IF OBJECT_ID('CalculateManagerBonus', 'P') IS NOT NULL
    DROP PROCEDURE CalculateManagerBonus
go
CREATE PROCEDURE CalculateManagerBonus
    @TargetAmount DECIMAL(10,2) = 100000
AS
BEGIN
    SELECT
        SaleMonth as 'Month',
        SUM(TotalAmount) as 'Actual revenue',
        @TargetAmount as 'Planned revenue',
        CASE
            WHEN SUM(TotalAmount) >= @TargetAmount THEN 'Plan fulfilled'
            ELSE 'Plan not fulfilled'
        END as 'Status',
        CASE
            WHEN SUM(TotalAmount) >= @TargetAmount THEN SUM(TotalAmount) * 0.05
            ELSE 0
        END as 'Manager bonus'
    FROM Sales
    GROUP BY SaleMonth
    ORDER BY SaleMonth
END
go
-- Demonstration
-- Sales analysis for Q1 2024
EXEC GetSalesReport '01.01.2024', '31.03.2024'
go
-- Calculating manager bonuses
EXEC CalculateManagerBonus @TargetAmount = 50000
go
```

```
-- Quick current sales analysis
SELECT
    'Current month: ' + FORMAT(GETDATE(), 'MMMM yyyy') as 'Period',
    (SELECT COUNT(*) FROM Sales WHERE SaleDate >= DATEFROMPARTS(YEAR(GETDATE()),
MONTH(GETDATE()), 1)) as 'Sales in month',
    (SELECT SUM(TotalAmount) FROM Sales WHERE SaleDate >=
DATEFROMPARTS(YEAR(GETDATE()), MONTH(GETDATE()), 1)) as 'Revenue in month'
go
-- Cleanup
IF OBJECT_ID('Sales', 'U') IS NOT NULL
    DROP TABLE Sales
go
IF OBJECT_ID('GetSalesReport', 'P') IS NOT NULL
    DROP PROCEDURE GetSalesReport
go
IF OBJECT_ID('CalculateManagerBonus', 'P') IS NOT NULL
    DROP PROCEDURE CalculateManagerBonus
go
```

Key technologies:

- Computed fields for automatic calculations
- Aggregate functions for data analysis
- Conditional logic for business rules
- Data formatting for reporting
- Parameterized procedures for flexible reports

3) PostgreSQL

Task:

Determine the TOP 5 products that brought the greatest profit for the last quarter, and also output information about average check, number of orders, and product name.

Tables:

- **ORDERS:** Order (ORDER_ID, CUSTOMER_ID, PRODUCT_ID, AMOUNT, ORDER_DATE)
- **PRODUCTS:** Products (PRODUCT_ID, NAME, PRICE)

Query:

```
CREATE TABLE PRODUCTS (  
    PRODUCT_ID SERIAL PRIMARY KEY,  
    NAME VARCHAR(100) NOT NULL,  
    PRICE DECIMAL(10, 2) NOT NULL  
);
```

```
CREATE TABLE ORDERS (  
    ORDER_ID SERIAL PRIMARY KEY,  
    CUSTOMER_ID INT NOT NULL,  
    PRODUCT_ID INT NOT NULL REFERENCES PRODUCTS(PRODUCT_ID),  
    AMOUNT DECIMAL(10, 2) NOT NULL,  
    ORDER_DATE TIMESTAMP NOT NULL  
);
```

```
INSERT INTO PRODUCTS (NAME, PRICE)  
VALUES  
    ('Smartphone', 35000.00),  
    ('Laptop', 60000.00),  
    ('Tablet', 25000.00),  
    ('Monitor', 15000.00),  
    ('Speakers', 5000.00);
```

```
INSERT INTO ORDERS (CUSTOMER_ID, PRODUCT_ID, AMOUNT, ORDER_DATE)  
VALUES  
    (101, 1, 35000.00, '2023-09-15 10:30:00'),  
    (102, 2, 60000.00, '2023-09-20 14:15:00'),  
    (103, 3, 25000.00, '2023-09-25 16:45:00'),  
    (104, 1, 35000.00, '2025-05-01 09:00:00'),  
    (105, 4, 15000.00, '2025-05-05 11:30:00'),  
    (106, 5, 5000.00, '2025-06-10 13:15:00'),  
    (107, 2, 60000.00, '2025-07-15 15:00:00'),  
    (108, 3, 25000.00, '2025-07-20 17:30:00');
```



```
WITH LastQuarterOrders AS (  
    -- Subquery to select orders from last 3 months  
    SELECT *  
    FROM ORDERS  
    WHERE ORDER_DATE >= (current_date - interval '3 months')  
) ,  
ProfitPerProduct AS (  
    -- Subquery to calculate profit per product  
    SELECT  
        *.PRODUCT_ID,  
        SUM(*.AMOUNT) AS TotalRevenue,  
        COUNT(*) AS OrderCount,  
        AVG(*.AMOUNT) AS AverageOrderAmount  
    FROM LastQuarterOrders *  
    GROUP BY *.PRODUCT_ID  
) ,  
FinalResults AS (  
    -- Subquery to join product name and final sorting  
    SELECT  
        P.NAME AS ProductName,  
        P.PRICE,  
        PPP.TotalRevenue,  
        PPP.OrderCount,  
        PPP.AverageOrderAmount  
    FROM ProfitPerProduct PPP  
    JOIN PRODUCTS P ON PPP.PRODUCT_ID = P.PRODUCT_ID  
    ORDER BY PPP.TotalRevenue DESC  
    LIMIT 5  
)  
SELECT  
    FR.ProductName,  
    FR.Price,  
    FR.TotalRevenue AS Revenue,  
    FR.OrderCount AS Orders,  
    ROUND(FR.AverageOrderAmount, 2) AS AverageOrder  
FROM FinalResults FR;
```

15. Built-in Packages, Functions, and Procedures of the Oracle Database Dialect

15.1. General Information

The DBMS contains a specific set of procedures, functions, and packages for the Oracle Database dialect.

DBMS packages (Database Management System Packages) are specialized software components designed to facilitate database management and extend DBMS functionality. Each package includes a set of procedures, functions, and variables intended to perform specific tasks, such as performance management, security, audit logging, and others.

Why are DBMS packages needed?

- **Simplifying administration:** simplifies performing daily database administration tasks;
- **Automating tasks:** allows automating regular operations, such as backup, audit log cleanup, etc;
- **Extending functionality:** enhances DBMS capabilities by providing new tools and features.

UTL packages (Utility Packages) are a set of predefined packages in the Oracle DBMS that provide utility functions for interacting with the operating system, network, files, and for performing other administrative tasks that cannot be implemented using pure SQL.

Why are UTL packages needed?

- **Working with external data:** allows reading and writing text files directly from the DB;
- **Network interaction:** provides the ability to send HTTP requests and work with network data;
- **Built-in mail:** gives the ability to send emails directly from the DBMS;
- **Service operations:** provides functions for data formatting, working with error codes, and other utilities.

PLVSTR (PL/Vision STRing) is a package from the PL/Vision library that provides an

extended set of utilities for working with strings, complementing standard Oracle SQL functions.

Why is PLVSTR needed?

- **Extended string analysis:** provides functions for searching substrings, counting occurrences, and more complex analysis.
- **Improved processing:** allows easily splitting strings into parts, removing extra characters, and performing complex manipulations.
- **Simplifying code:** encapsulates complex string logic into simple-to-call procedures and functions.

PLVDATE (PL/Vision DATE) is a package from the PL/Vision library that extends the capabilities of standard date handling in Oracle.

Why is PLVDATE needed?

- **Extended date arithmetic:** simplifies date calculations (e.g., "last day of previous month").
- **Intuitive checks:** provides functions for easy checking, such as whether a date is a business day, and for formatting output.
- **Simplifying time intervals:** offers convenient methods for working with time intervals and calendar.

Supported procedures, functions, packages of the **Oracle Database** dialect:

1. DBMS_OUTPUT.PUT
2. DBMS_OUTPUT.PUT_LINE
3. DBMS_OUTPUT.ENABLE
4. DBMS_OUTPUT.DISABLE
5. DBMS_OUTPUT.GET_LINE
6. DBMS_OUTPUT.GET_LINES
7. DBMS_ASSERT.NOOP
8. DBMS_ASSERT.ENQUOTE_NAME
9. DBMS_ASSERT.ENQUOTE_LITERAL
10. DBMS_ASSERT.SIMPLE_SQL_NAME

11. DBMS_ASSERT.QUALIFIED_SQL_NAME
12. DBMS_RANDOM.INITIALIZE
13. DBMS_RANDOM.TERMINATE
14. DBMS_RANDOM.NORMAL
15. DBMS_RANDOM.RANDOM
16. DBMS_RANDOM.SEED
17. DBMS_RANDOM.VALUE
18. UTL_CALL_STACK.OWNER
19. UTL_CALL_STACK.CURRENT_EDITION
20. UTL_CALL_STACK.BACKTRACE_DEPTH
21. UTL_CALL_STACK.ERROR_DEPTH
22. UTL_CALL_STACK.DYNAMIC_DEPTH
23. UTL_CALL_STACK.LEXICAL_DEPTH
24. UTL_CALL_STACK.ERROR_NUMBER
25. UTL_CALL_STACK.BACKTRACE_UNIT
26. UTL_CALL_STACK.ERROR_MSG
27. UTL_ENCODE.BASE64_ENCODE
28. UTL_ENCODE.BASE64_DECODE
29. UTL_ENCODE.UUENCODE
30. UTL_ENCODE.TEXT_DECODE
31. UTL_ENCODE.QUOTED_PRINTABLE_DECODE
32. UTL_ENCODE.QUOTED_PRINTABLE_ENCODE
33. UTL_ENCODE.MIMEHEADER_DECODE
34. UTL_I18N.STRING_TO_RAW
35. UTL_I18N.RAW_TO_CHAR
36. UTL_I18N.VALIDATE_CHARACTER_ENCODING
37. UTL_I18N.ESCAPE_REFERENCE
38. UTL_I18N.UNESCAPE_REFERENCE
39. UTL_I18N.GET_DEFAULT_CHARSET

40. UTL_I18N.GET_DEFAULT_ISO_CURRENCY
41. UTL_I18N.GET_DEFAULT_LINGUISTIC_SORT
42. UTL_I18N.GET_MAX_CHARACTER_SIZE
43. UTL_I18N.GET_TRANSLATION
44. UTL_I18N.MAP_CHARSET
45. UTL_I18N.MAP_FROM_SHORT_LANGUAGE
46. UTL_I18N.MAP_LANGUAGE_FROM_ISO
47. UTL_I18N.MAP_TO_SHORT_LANGUAGE
48. UTL_I18N.TRANSLITERATE
49. UTL_I18N.MAP_LOCALE_TO_ISO
50. UTL_IDENT.IS_ORACLE_SERVER
51. UTL_IDENT.IS_ORACLE_CLIENT
52. UTL_IDENT.IS_TIMESTEN
53. UTL_IDENT.IS_ORACLE_FORMS
54. UTL_INADDR.GET_HOST_NAME;
55. UTL_INADDR.GET_HOST_ADDRESS;
56. UTL_LMS.FORMAT_MESSAGE
57. UTL_LMS.GET_MESSAGE
58. UTL_MATCH.EDIT_DISTANCE
59. UTL_MATCH.EDIT_DISTANCE_SIMILARITY
60. UTL_MATCH.JARO_WINKLER
61. UTL_MATCH.JARO_WINKLER_SIMILARITY
62. UTL_RAW.BIT_AND
63. UTL_RAW.BIT_OR
64. UTL_RAW.BIT_XOR
65. UTL_RAW.BIT_COMPLEMENT
66. UTL_RAW.CAST_FROM_BINARY_DOUBLE
67. UTL_RAW.CAST_TO_BINARY_DOUBLE
68. UTL_RAW.CAST_FROM_BINARY_FLOAT

- 69. UTL_RAW.CAST_TO_BINARY_FLOAT
- 70. UTL_RAW.CAST_FROM_BINARY_INTEGER
- 71. UTL_RAW.CAST_TO_BINARY_INTEGER
- 72. UTL_RAW.CAST_TO_VARCHAR2
- 73. UTL_RAW.CAST_TO_RAW
- 74. UTL_RAW.COMPARE
- 75. UTL_RAW.CONCAT
- 76. UTL_RAW.COPIES
- 77. UTL_RAW.LENGTH
- 78. UTL_RAW.OVERLAY
- 79. UTL_RAW.REVERSE
- 80. UTL_RAW.SUBSTR
- 81. UTL_RAW.TRANSLATE
- 82. UTL_RAW.TRANSLITERATE
- 83. UTL_RAW.XRANGE
- 84. UTL_SMTP.OPEN_CONNECTION
- 85. UTL_SMTP.CLOSE_CONNECTION
- 86. UTL_SMTP.QUIT
- 87. UTL_SMTP.NOOP
- 88. UTL_SMTP.HELO
- 89. UTL_SMTP.EHLO
- 90. UTL_SMTP.MAIL
- 91. UTL_SMTP.RCPT
- 92. UTL_SMTP.RSET
- 93. UTL_SMTP.DATA
- 94. UTL_SMTP.OPEN_DATA
- 95. UTL_SMTP.WRITE_DATA
- 96. UTL_SMTP.CLOSE_DATA
- 97. UTL_TCP.OPEN_CONNECTION

- 98. UTL_TCP.CLOSE_CONNECTION
- 99. UTL_TCP.WRITE_TEXT
- 100. UTL_TCP.GET_TEXT
- 101. UTL_TCP.FLUSH
- 102. UTL_TCP.AVAILABLE
- 103. UTL_TCP.WRITE_LINE
- 104. UTL_TCP.GET_LINE
- 105. UTL_TCP.WRITE_RAW
- 106. UTL_TCP.GET_RAW
- 107. UTL_TCP.CLOSE_ALL_CONNECTIONS
- 108. UTL_MAIL.SEND
- 109. UTL_MAIL.SEND_ATTACH_VARCHAR2
- 110. UTL_MAIL.SEND_ATTACH_RAW
- 111. UTL_URL.ESCAPE;
- 112. UTL_URL.UNESCAPE;
- 113. PLVDATE.PREV_BIZDAY
- 114. PLVDATE.ISBIZDAY
- 115. PLVDATE.SET_NONBIZDAY
- 116. PLVDATE.UNSET_NONBIZDAY
- 117. PLVDATE.BIZDAYS_BETWEEN
- 118. PLVDATE.ADD_BIZDAYS
- 119. PLVDATE.NEAREST_BIZDAY
- 120. PLVDATE.NEXT_BIZDAY
- 121. PLVDATE.USE_EASTER
- 122. PLVDATE.UNUSE_EASTER
- 123. PLVDATE.USING_EASTER
- 124. PLVDATE.USE_GREAT_FRIDAY
- 125. PLVDATE.INCLUDE_START
- 126. PLVDATE.NOINCLUDE_START

- 127. PLVDATE.INCLUDING_START
 - 128. PLVDATE.DEFAULT_HOLIDAYS
 - 129. PLVDATE.UNUSE_GREAT_FRIDAY
 - 130. PLVDATE.USING_GREAT_FRIDAY
 - 131. PLVDATE.DAYS_INMONTH
 - 132. PLVDATE.ISLEAPYEAR
 - 133. PLVSTR.IS_PREFIX
 - 134. PLVSTR.SUBSTR
 - 135. PLVSTR.INSTR
 - 136. PLVSTR.NORMALIZE
 - 137. PLVSTR.LPART
 - 138. PLVSTR.RPART
 - 139. PLVSTR.LSTRIP
 - 140. PLVSTR.RSTRIP
 - 141. PLVSTR.RIGHT
 - 142. PLVSTR.SWAP
 - 143. PLVSTR.LEFT
 - 144. PLVSTR.RVRS
 - 145. PLVSTR.BETWN
-
- 1. NLS_CHARSET_ID
 - 2. NLS_CHARSET_NAME
 - 3. ADD_MONTHS
 - 4. LAST_DAY
 - 5. LENGTH
 - 6. LNNVL
 - 7. MONTHS_BETWEEN
 - 8. NEXT_DAY
 - 9. NLSSORT
 - 10. NVL

11. NVL2
12. ROUND
13. SESSIONTIMEZONE
14. SUBSTR
15. SYS_EXTRACT_UTC
16. SYSDATE FROM DUAL
17. TO_CHAR
18. TO_DATE
19. TO_MULTI_BYTE
20. TO_SINGLE_BYTE
21. TRUNC

15.2. Description of Package Objects

15.2.1. DBMS_OUTPUT Package

DBMS_OUTPUT.PUT

The procedure sends part of a string to the output buffer without adding a newline character. Text accumulates in the buffer until PUT_LINE or NEW_LINE is called.

Input parameters:

- item (VARCHAR2) – string to output to the buffer.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Gradually forming long output strings with multiple calls.
- Outputting text without line break for subsequent addition.
- Creating custom output formats with element separation.

Query examples:

```
SET SERVEROUTPUT ON
BEGIN
DBMS_OUTPUT.PUT('Hello');
DBMS_OUTPUT.PUT(' ');
DBMS_OUTPUT.PUT('World!');
DBMS_OUTPUT.NEW_LINE;
END;
/
-- Result:
--Hello World!
```

```
SET SERVEROUTPUT ON
BEGIN
DBMS_OUTPUT.PUT('Current time: ');
DBMS_OUTPUT.PUT(TO_CHAR(SYSDATE, 'HH24:MI:SS'));
DBMS_OUTPUT.NEW_LINE;
END;
/
```

DBMS_OUTPUT.PUT_LINE

The function sends a string to the output buffer, from where it can be displayed by SQL*Plus or similar tools.

Input parameters:

- item (VARCHAR2) – string to output to the buffer.

Output parameters:

- The function does not return a value.

Typical usage scenarios:

- Debugging stored procedures and triggers by outputting intermediate messages.
- Outputting final results of queries or calculations.
- Reporting errors and warnings to the developer.

Query examples:

```
SET SERVEROUTPUT ON
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello World!');
END;
/
□- Result:
--Hello World!
```

DBMS_OUTPUT.DISABLE

The function disables message output to the buffer and clears the buffer of accumulated data. After disabling, any subsequent calls to PUT, PUT_LINE, and similar procedures are ignored until output is re-enabled with the ENABLE command.

Input parameters:

- No parameters.

Output parameters:

- The function does not return a value.

Typical usage scenarios:

- Ending a debug message output session after completing procedure testing.
- Clearing the output buffer to avoid unnecessary data in subsequent operations.

Query examples:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Buffer temporarily disabled.');
```

DBMS_OUTPUT.DISABLE(); -- Disable output

```
END;
/
□- Result:
--Buffer temporarily disabled.
```

DBMS_OUTPUT.GET_LINE

The function retrieves one line from the output buffer where messages were sent by PUT and PUT_LINE procedures. This function is useful for reading information from the buffer line by line and subsequently displaying it.

Input parameters:

- line (VARCHAR2) – variable in which the extracted line from the buffer is saved.
- status (NUMBER) – execution status:
- 0 — operation successful.
- 1 — no more lines in buffer to extract.

Output parameters:

- The function does not return a value.

Typical usage scenarios:

- Line-by-line reading of debug information from the buffer after executing a complex stored procedure.
- Extracting query results or logging for analysis or documentation.

Query examples:

```
DECLARE
  v_line VARCHAR2(32767);
  v_status INTEGER;
BEGIN
  DBMS_OUTPUT.PUT_LINE('First line');
  DBMS_OUTPUT.PUT_LINE('Second line');
  DBMS_OUTPUT.GET_LINE(v_line, v_status);
  DBMS_OUTPUT.PUT_LINE('Retrieved line: ' || v_line);
END;
/
□ Result:
--Retrieved line: First line
```

DBMS_OUTPUT.GET_LINES

The function retrieves an array of lines from the output buffer where messages were sent by the PUT_LINE procedure. Used for efficient bulk reading of multiple lines from the buffer simultaneously.

Input parameters:

- lines (PL/SQL TABLE) – collection of lines (type CHARARR) in which extracted lines from the buffer are saved.

- numlines (NUMBER) – number of lines requested. The parameter value is updated by the function and shows the actual number of lines received.

Output parameters:

- The function does not return a value. *Typical usage scenarios:*
- Bulk retrieval of debug information from the buffer after completing long operations.
- Optimized extraction of a large number of lines for subsequent processing or display.

15.2.2. DBMS_ASSERT Package

DBMS_ASSERT.NOOP

The function returns the input string unchanged. Used as a stub for validating SQL expressions without actual checking.

Input parameters:

- input_string (VARCHAR2) – original string to be returned unchanged.

Output parameters:

- Returns the original string unmodified.

Typical usage scenarios:

- Temporary stub during development to bypass checks.
- Testing application logic without real validation.
- Debugging call chains.
- Replacing other DBMS_ASSERT functions in test scenarios.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
v_sql VARCHAR2(1000);
BEGIN
v_sql := DBMS_ASSERT.NOOP('abc');
DBMS_OUTPUT.PUT_LINE(v_sql);
END;
/
-- Result:
-- abc
```

DBMS_ASSERT.ENQUOTE_NAME

The function encloses a database object name in quotes to form a correct SQL identifier.

Input parameters:

- name (VARCHAR2) – database object name to enclose in quotes.
- capitalize (BOOLEAN) – flag indicating name conversion to uppercase (TRUE - convert, FALSE - leave as is).

Output parameters:

- Returns a string with the object name enclosed in double quotes.

Typical usage scenarios:

- Dynamic formation of SQL queries with object names.
- Ensuring correctness of identifiers in DDL operations.
- Working with object names containing special characters or in lower case.
- Preparing safe SQL expressions for execution.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
v_sql VARCHAR2(1000);
BEGIN
v_sql := DBMS_ASSERT.ENQUOTE_NAME('abc', false);
DBMS_OUTPUT.PUT_LINE(v_sql);
END;
/
-- Result:
-- "abc"
```

DBMS_ASSERT.ENQUOTE_LITERAL

The function encloses a string literal in single quotes for safe use in SQL queries.

Input parameters:

- str (VARCHAR2) – string literal to enclose in quotes.

Output parameters:

- Returns the string enclosed in single quotes.

Typical usage scenarios:

- Safe formation of SQL queries with string literals.
- Preventing SQL injections when dynamically building queries.
- Preparing values for WHERE conditions in SQL expressions.
- Processing user input before inserting into queries.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
v_sql VARCHAR2(1000);
BEGIN
v_sql := DBMS_ASSERT.ENQUOTE_LITERAL('World');
DBMS_OUTPUT.PUT_LINE(v_sql);
END;
/
-- Result:
-- 'World'
```

DBMS_ASSERT.SIMPLE_SQL_NAME

The function checks if the input string is a valid simple SQL identifier and returns it unchanged if the check passes.

Input parameters:

- name (VARCHAR2) – string to check for compliance with SQL identifier naming rules.

Output parameters:

- Returns the original string if it is a valid SQL identifier.

Typical usage scenarios:

- Validation of table, column, and other database object names.
- Checking user input before using in dynamic SQL.
- Preventing SQL injections through object names.
- Ensuring correctness of formed SQL queries.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
v_sql VARCHAR2(1000);
BEGIN
v_sql := DBMS_ASSERT.SIMPLE_SQL_NAME('abc_efg');
DBMS_OUTPUT.PUT_LINE(v_sql);
END;
/
```

DBMS_ASSERT.QUALIFIED_SQL_NAME

The function checks if the input string is a valid qualified SQL name (consisting of multiple parts separated by dots) and returns it unchanged if the check passes.

Input parameters:

- name (VARCHAR2) – string to check for compliance with qualified SQL identifier naming rules.

Output parameters:

- Returns the original string if it is a valid qualified SQL name.

Typical usage scenarios:

- Validation of full paths to database objects (e.g., schema.table.column).
- Checking complex identifiers before using in dynamic SQL.
- Ensuring security when working with object names containing special characters.
- Preparing correct references to objects in SQL queries.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
v_sql VARCHAR2(1000);
BEGIN
v_sql := DBMS_ASSERT.QUALIFIED_SQL_NAME('abc_efg."Tr34$6-".j@sa');
DBMS_OUTPUT.PUT_LINE(v_sql);
END;
/
-- Result:
-- abc_efg."Tr34$6-".j@sa
```

15.2.3. DBMS_RANDOM Package

DBMS_RANDOM.INITIALIZE

The procedure initializes the random number generator with a given integer seed value.

Input parameters:

- seed (BINARY_INTEGER) – integer seed value for initializing the generator.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Setting a fixed seed value for reproducible random number sequences (e.g., in tests

or demos).

- Synchronizing generator state across different sessions (when using the same seed).

Query examples:

```
DECLARE
  num NUMBER;
BEGIN
  DBMS_RANDOM.INITIALIZE(5);
  num := DBMS_RANDOM.RANDOM();
  DBMS_OUTPUT.PUT_LINE('NUM: ' || num);
  DBMS_RANDOM.TERMINATE();
END;
/
```

DBMS_RANDOM.TERMINATE

The procedure explicitly terminates the random number generator.

Input parameters:

- No parameters.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Explicitly freeing generator resources.

Query examples:

```
DECLARE
  num NUMBER;
BEGIN
  DBMS_RANDOM.INITIALIZE(5);
  num := DBMS_RANDOM.RANDOM();
  DBMS_OUTPUT.PUT_LINE('NUM: ' || num);
  DBMS_RANDOM.TERMINATE();
END;
/
```

DBMS_RANDOM.NORMAL

The function returns a random number having a normal (Gaussian) distribution with a given mean value (mean) and standard deviation (sigma).

Input parameters:

- No parameters.

Output parameters:

- NUMBER – random number with normal distribution.

Typical usage scenarios:

- Generating realistic test data simulating natural processes (height, weight, measurement errors).
- Modeling financial indicators, market volatility.
- Creating data for testing statistical algorithms and reports.
- Simulating noise or random deviations in scientific calculations.

Query examples:

```
DECLARE
  num NUMBER;
BEGIN
  num := DBMS_RANDOM.NORMAL();
  DBMS_OUTPUT.PUT_LINE('NUM: ' || num);
END;
/
```

DBMS_RANDOM.RANDOM

The function returns a random integer in the range from -2^{31} to 2^{31} . Deprecated function, use DBMS_RANDOM.VALUE or DBMS_RANDOM.STRING is recommended.

Input parameters:

No parameters.

Output parameters:

- **BINARY_INTEGER** – random integer.

Typical usage scenarios:

- Getting a random integer in legacy code.

Query examples:

```
DECLARE
  num NUMBER;
BEGIN
  DBMS_RANDOM.INITIALIZE(5);
  num := DBMS_RANDOM.RANDOM();
  DBMS_OUTPUT.PUT_LINE('NUM: ' || num);
  DBMS_RANDOM.TERMINATE();
END;
/
```

DBMS_RANDOM.SEED

The procedure sets the seed value for the random number generator. Used to create a reproducible sequence of "random" numbers. Can accept either a numeric or string value.

Input parameters:

- seed (PLS_INTEGER | VARCHAR2) – seed value. Can be an integer or string.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Reproducing the same sequence of random numbers for debugging and testing purposes.
- Guaranteeing identical data generation in different sessions with the same seed.
- Initializing the generator with a more random value (e.g., using a string or current time in microseconds).

Query examples:

```
DECLARE
  num NUMBER;
BEGIN
  DBMS_RANDOM.SEED(10);
  num := DBMS_RANDOM.RANDOM();
  DBMS_OUTPUT.PUT_LINE('NUM: ' || num);
END;
/
```

```
DECLARE  num  NUMBER;  BEGIN  DBMS_RANDOM.SEED('a');  num  :=
DBMS_RANDOM.RANDOM(); DBMS_OUTPUT.PUT_LINE('NUM: ' || num); END; /
```

DBMS_RANDOM.STRING

The function generates a random string of a given type and length.

Input parameters:

- opt (CHAR) – type of characters to generate in the string.
- len (NUMBER) – length of the generated string.

Output parameters:

- VARCHAR2 – random string of given length and type.

Typical usage scenarios:

- Generating temporary passwords, access tokens, one-time codes.
- Creating unique session or transaction identifiers.
- Filling test data for string fields in tables.
- Generating random filenames, directories.
- Creating test data for testing string processing of different formats.

_Query examples:

```
SELECT DBMS_RANDOM.STRING('A', 30);
```

```
SELECT DBMS_RANDOM.STRING('U', 30);
```

```
SELECT DBMS_RANDOM.STRING('L', 30);
```

```
SELECT DBMS_RANDOM.STRING('X', 30);
```

```
SELECT DBMS_RANDOM.STRING('P', 30);
```

DBMS_RANDOM.VALUE

The function returns a random number of type NUMBER. Can be called without parameters or with a specified range.

Input parameters:

- low (NUMBER) – lower bound of the range (inclusive). Default is 0.
- high (NUMBER) – upper bound of the range (exclusive). Default is 1.

Output parameters:

- NUMBER – random number within the range.

Typical usage scenarios:

- Generating random fractional numbers for simulations, games, test data.
- Creating random percentages, fractions, probabilities.
- Getting random order of records in a selection (e.g., ORDER BY DBMS_RANDOM.VALUE).
- Modeling random time intervals or delays.
- Forming random values for filling fields for demonstration purposes.

Query examples:

```
DECLARE
  num NUMBER;
BEGIN
  num := DBMS_RANDOM.VALUE();
  DBMS_OUTPUT.PUT_LINE('NUM: ' || num);
END;
/
```

```
DECLARE num NUMBER; BEGIN num := DBMS_RANDOM.VALUE(10, 20);
DBMS_OUTPUT.PUT_LINE('NUM: ' || num); END; /
```

15.2.4. UTL_CALL_STACK Package

UTL_CALL_STACK.OWNER

The function returns the name of the object (schema) owner where the program unit resides at the specified depth in the call stack.

Input parameters:

- depth (PLS_INTEGER) – depth in the call stack (default 0 – current unit).

Output parameters:

- Returns the schema owner name.

Typical usage scenarios:

- Debugging complex chains of procedure and function calls.
- Audit and logging to determine the source of a call.
- Dynamic decision-making depending on execution context (e.g., checking permissions).

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
    v := UTL_CALL_STACK.OWNER(1);
    DBMS_OUTPUT.PUT_LINE('OWNER(1)= ' || v);
END;
/
```

UTL_CALL_STACK.CURRENT_EDITION

The function returns the name of the current (active) edition of the database, in the context of which the code is executed.

Input parameters:

- No parameters.

Output parameters:

- Returns the name of the current edition.

Typical usage scenarios:

- Checking execution context in environments using editions for application updates (Edition-Based Redefinition).
- Logging and debugging in multi-edition environments.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  v := UTL_CALL_STACK.CURRENT_EDITION(1);
  DBMS_OUTPUT.PUT_LINE('CURRENT_EDITION(1)= ' || v);
END;
/
```

UTL_CALL_STACK.BACKTRACE_DEPTH

The function returns the number of available entries in the stack backtrace. Backtrace contains the sequence of calls that led to the current execution point, including calls not related to PL/SQL.

Input parameters:

- No parameters.

Output parameters:

- Returns the backtrace depth.

Typical usage scenarios:

- Getting a complete picture of the call stack, including calls from outside PL/SQL.
- Creating detailed error logs with full trace.
- Diagnosing complex interactions between components.

Query examples:


```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  v := UTL_CALL_STACK.BACKTRACE_DEPTH();
  DBMS_OUTPUT.PUT_LINE('BACKTRACE_DEPTH=' || v);
END;
/
```

UTL_CALL_STACK.ERROR_DEPTH

The function returns the number of entries in the call stack at the moment the last exception occurred. This depth corresponds to the context in which the exception was raised.

Input parameters:

- No parameters.

Output parameters:

- Returns the stack depth at the error point.

Typical usage scenarios:

- Analyzing the call stack inside exception blocks (WHEN OTHERS) to determine the error source.
- Constructing custom error messages indicating the location of occurrence.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  v := UTL_CALL_STACK.ERROR_DEPTH();
  DBMS_OUTPUT.PUT_LINE('ERROR_DEPTH=' || v);
END;
/
```

UTL_CALL_STACK.DYNAMIC_DEPTH

The function returns the current depth of the dynamic call stack. The dynamic stack includes calls made through dynamic SQL (e.g., EXECUTE IMMEDIATE).

Input parameters:

- No parameters.

Output parameters:

- Returns the dynamic stack depth.

Typical usage scenarios:

- Debugging code that uses dynamic SQL.
- Understanding the nesting of calls generated at runtime.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  v := UTL_CALL_STACK.DYNAMIC_DEPTH();
  DBMS_OUTPUT.PUT_LINE('DYNAMIC_DEPTH=' || v);
END;
/
```

UTL_CALL_STACK.LEXICAL_DEPTH

The function returns the lexical depth (nesting level) of a subroutine (procedure or function) in the specified program unit at the given stack depth.

Input parameters:

- depth (PLS_INTEGER) – depth in the call stack (default 0 – current unit).

Output parameters:

- Returns the lexical depth (0 for outer subroutine, 1 for nested, etc.).

Typical usage scenarios:

- Determining context in nested subroutines.
- Checking code structure during refactoring or analysis.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  v := UTL_CALL_STACK.LEXICAL_DEPTH(1);
  DBMS_OUTPUT.PUT_LINE('LEXICAL_DEPTH(1)= ' || v);
END;
/
```

UTL_CALL_STACK.ERROR_NUMBER

The function returns the error number of the last exception that occurred in the session.

Input parameters:

- No parameters.

Output parameters:

- Returns the Oracle error number (e.g., -6502 for VALUE_ERROR).

Typical usage scenarios:

- Getting the error code in a WHEN OTHERS block for subsequent processing or logging.
- Integration with monitoring and alerting systems.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  IF UTL_CALL_STACK.ERROR_DEPTH() > 0 THEN
    v := UTL_CALL_STACK.ERROR_NUMBER(1);
  ELSE
    v := 'no errors';
  END IF;
  DBMS_OUTPUT.PUT_LINE('ERROR_NUMBER(1)= ' || v);
END;
/
```

UTL_CALL_STACK.BACKTRACE_UNIT

The function returns the name of the program unit at the specified depth in the stack backtrace.

Input parameters:

- `backtrace_depth` (PLS_INTEGER) – depth in the stack backtrace.

Output parameters:

- Returns the program unit name (e.g., package, procedure, function name).

Typical usage scenarios:

- Building a detailed stack trace for logging or displaying to the user.
- Analyzing the sequence of calls leading to an error.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  IF UTL_CALL_STACK.BACKTRACE_DEPTH() > 0 THEN
    v := UTL_CALL_STACK.BACKTRACE_UNIT(1);
  ELSE
    v := 'no backtrace';
  END IF;
  DBMS_OUTPUT.PUT_LINE('BACKTRACE_UNIT(1)= ' || v);
END;
/
```

UTL_CALL_STACK.ERROR_MSG

The function returns the text message of the last exception that occurred in the session.

Input parameters:

- No parameters.

Output parameters:

- Returns the error message text.

Typical usage scenarios:

- Logging understandable error messages.
- Forming user notifications with details of the original error.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE v VARCHAR2(4000);
BEGIN
  IF UTL_CALL_STACK.ERROR_DEPTH() > 0 THEN
    v := UTL_CALL_STACK.ERROR_MSG(1);
  ELSE
    v := 'no errors';
  END IF;
  DBMS_OUTPUT.PUT_LINE('ERROR_MSG(1)= ' || v);
END;
/
```

15.2.5. UTL_ENCODE Package

UTL_ENCODE.BASE64_ENCODE

The function converts binary data (RAW) to a Base64-encoded string.

Input parameters:

- r (RAW) – source binary data to encode.
- charset_id (PLS_INTEGER) – character set identifier for the output string (optional).
- newline (PLS_INTEGER) – flag for inserting newline characters every 76 characters (optional, e.g., CHR(13) || CHR(10)).

Output parameters:

- Returns encoded data in Base64 format.

Typical usage scenarios:

- Preparing binary data (e.g., files, signatures) for transmission through text protocols (HTTP, email).
- Encoding attachments for emails sent via UTL_MAIL.
- Storing binary data in text fields (CLOB, VARCHAR2).

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(2000);
BEGIN
  v := UTL_ENCODE.BASE64_ENCODE('48656C6C6F2E');
  DBMS_OUTPUT.PUT_LINE('BASE64 (HEX ASCII) = ' || RAWTOHEX(v));
END;
/
```

UTL_ENCODE.BASE64_DECODE

The function decodes a string from Base64 format back to original binary data (RAW).

Input parameters:

- r (RAW) – Base64-encoded data to decode.

Output parameters:

- Returns decoded binary data.

Typical usage scenarios:

- Processing email attachments received in Base64 encoding.
- Extracting binary data stored in text form.
- Working with APIs that transmit binary content in Base64.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(2000);
BEGIN
  v := UTL_ENCODE.BASE64_DECODE('5347567362473875'); -- "SGVsbG8u"
  DBMS_OUTPUT.PUT_LINE('DECODED HEX = ' || RAWTOHEX(v));      -- 48656C6C6F2E
END;
/
```

UTL_ENCODE.UUENCODE

The function encodes binary data (RAW) into UUENCODE format.

Input parameters:

- r (RAW) – source binary data to encode.
- type (PLS_INTEGER) – encoding type (e.g., to specify filename and access mode in the header).

Output parameters:

- Returns encoded data in UUENCODE format.

Typical usage scenarios:

- Processing legacy systems or data where UUENCODE is still used.
- Compatibility with protocols historically using this encoding.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  u RAW(32767);
  v RAW(32767);
BEGIN
  u := UTL_ENCODE.UUENCODE('48656C6C6F0A');
  v := UTL_ENCODE.UUDECODE(u);
  DBMS_OUTPUT.PUT_LINE('ROUNDRIP HEX = ' || RAWTOHEX(v)); -- 48656C6C6F0A
END;
/
```

UTL_ENCODE.TEXT_DECODE

The function decodes text from a given character set into RAW.

Input parameters:

- buf (VARCHAR2) – text to decode.
- charset (VARCHAR2) – source character set of the text.

Output parameters:

- Returns decoded binary data.

Typical usage scenarios:

- Converting text data received from external sources with various encodings into a unified internal format.
- Preparing text for cryptographic operations that require RAW data.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  s VARCHAR2(4000);
BEGIN
  s := UTL_ENCODE.TEXT_DECODE('0J/RgNC40LLQtdGC', 'UTF8', 1);
  DBMS_OUTPUT.PUT_LINE('DECODED = ' || s); -- Hello
END;
/
```

UTL_ENCODE.QUOTED_PRINTABLE_DECODE

The function decodes data from Quoted-Printable format back to original RAW.

Input parameters:

- buf (RAW) – data in Quoted-Printable encoding.

Output parameters:

- Returns decoded binary data.

Typical usage scenarios:

- Processing email content encoded with the Quoted-Printable method.
- Decoding text data containing non-ASCII characters.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(32767);
BEGIN
  v := UTL_ENCODE.QUOTED_PRINTABLE_DECODE('48656C6C6F203D45323D38323D4143');
  DBMS_OUTPUT.PUT_LINE('DECODED HEX = ' || RAWTOHEX(v)); -- 48656C6C6F20E282AC
END;
/
```


UTL_ENCODE.QUOTED_PRINTABLE_ENCODE

The function encodes binary data (RAW) into Quoted-Printable format.

Input parameters:

- buf (RAW) – source binary data to encode.

Output parameters:

- Returns encoded data in Quoted-Printable format.

Typical usage scenarios:

- Preparing email text for correct transmission via SMTP, especially for non-ASCII characters.
- Encoding data before inserting into mail headers.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(32767);
BEGIN
  v := UTL_ENCODE.QUOTED_PRINTABLE_ENCODE('48656C6C6F20E282AC');
  DBMS_OUTPUT.PUT_LINE('QP (HEX ASCII) = ' || RAWTOHEX(v)); --
48656C6C6F203D45323D38323D4143
END;
/
```

UTL_ENCODE.MIMEHEADER_DECODE

The function decodes a string from an email MIME header (e.g., Subject, From), which may be encoded to support non-ASCII characters.

Input parameters:

- buf (VARCHAR2) – encoded header string.

Output parameters:

- Returns decoded string.

Typical usage scenarios:

- Correctly displaying email subjects and sender names in different languages.
- Parsing incoming emails processed by Oracle tools.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  s VARCHAR2(4000);
BEGIN
  s := UTL_ENCODE.MIMEHEADER_DECODE('=?UTF-8?B?0J/RgNC40LLQtd6C?=' );
  DBMS_OUTPUT.PUT_LINE('DECODED: ' || s); -- Hello
END;
/
```

15.2.6. UTL_I18N Package

UTL_I18N.STRING_TO_RAW

The function converts a string (VARCHAR2) to RAW data type using the specified character set.

Input parameters:

str (VARCHAR2) – source string.

charset (VARCHAR2) – target character set (e.g., 'AL32UTF8', 'CL8MSWIN1251').

Output parameters:

- Returns binary representation of the string in the given encoding.

Typical usage scenarios:

- Preparing string data for cryptographic functions (e.g., DBMS_CRYPTO) that work with RAW.
- Converting text before sending to external systems with a specific encoding.
- Fixing string encoding for subsequent unambiguous interpretation.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_I18N.STRING_TO_RAW('é', 'WE8ISO8859P1');
  DBMS_OUTPUT.PUT_LINE('RAW=' || RAWTOHEX(r));
END;
/
```

UTL_I18N.RAW_TO_CHAR

The function converts binary data (RAW) to a string (VARCHAR2) using the specified character set.

Input parameters:

- r (RAW) – binary data.
- charset (VARCHAR2) – character set in which to interpret the RAW data.

Output parameters:

- Returns the decoded string.

Typical usage scenarios:

- Converting results of cryptographic operations (encryption, hashing) back to readable text.
- Reading text data received from external sources as a binary stream with a known encoding.
- Correctly displaying multilingual content.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(4000);
BEGIN
  v := UTL_I18N.RAW_TO_CHAR(HEXTORAW('616263646566C2AA'), 'utf8');
  DBMS_OUTPUT.PUT_LINE('VAL=' || v);
END;
/
```

UTL_I18N.VALIDATE_CHARACTER_ENCODING

The procedure checks whether a string can be correctly converted to the specified character set.

Input parameters:

- str (VARCHAR2) – string to validate.
- charset (VARCHAR2) – character set for validation.

Output parameters:

- The procedure does not return a value. If conversion is impossible, an exception is raised.

Typical usage scenarios:

- Pre-validating data before attempting conversion or saving to a system with a different encoding.
- Ensuring data integrity during migration or export/import.
- Checking user input against a required encoding.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  offs PLS_INTEGER;
BEGIN
  offs := UTL_I18N.VALIDATE_CHARACTER_ENCODING('hello é');
  DBMS_OUTPUT.PUT_LINE('OFFS=' || NVL(TO_CHAR(offs), 'NULL'));
END;
/
```

UTL_I18N.ESCAPE_REFERENCE

The function converts a string to a Unicode escape sequence (e.g., \u0410 for Cyrillic 'А').

Input parameters:

- str (VARCHAR2) – source string.
- charset (VARCHAR2) – encoding to convert escape sequences to (optional).

Output parameters:

- Returns a string with escape sequences.

Typical usage scenarios:

- Preparing strings for inclusion in XML, JSON, or other data formats where escape encoding of special or non-ASCII characters is required.
- Safely transmitting strings containing characters that may be misinterpreted by a protocol or application.
- Storing complex text in a simplified text environment.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(4000);
BEGIN
  v := UTL_I18N.ESCAPE_REFERENCE('A < B', 'US7ASCII');
  DBMS_OUTPUT.PUT_LINE('VAL=' || v);
END;
/
```

UTL_I18N.UNESCAPE_REFERENCE

The function converts escape sequences in a string (e.g., \u0410) back to corresponding characters.

Input parameters:

- str (VARCHAR2) – string containing escape sequences.

Output parameters:

- Returns string with restored characters.

Typical usage scenarios:

- Processing data received from external systems (web services, APIs) that use escape encoding.
- Reading and interpreting strings from configuration files or templates containing escape sequences.

- Decoding text after operations that used ESCAPE_REFERENCE.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(4000);
BEGIN
  v := UTL_I18N.UNESCAPE_REFERENCE('hello &#x41;');
  DBMS_OUTPUT.PUT_LINE('VAL=' || v);
END;
/
```

UTL_I18N.GET_DEFAULT_CHARSET

The function returns the name of the default character set for the current session or database.

Input parameters:

- type (PLS_INTEGER) – type of default encoding requested (e.g., for session, for database).

Output parameters:

- Returns the default encoding name (e.g., 'AL32UTF8').

Typical usage scenarios:

- Determining current encoding settings for correct processing of text data.
- Logging or auditing runtime environment parameters.
- Making decisions about the need for data conversion.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c1 VARCHAR2(30);
  c2 VARCHAR2(30);
  c3 VARCHAR2(30);
BEGIN
  c1 := UTL_I18N.GET_DEFAULT_CHARSET('French', 1, FALSE);
  c2 := UTL_I18N.GET_DEFAULT_CHARSET('French', 1, TRUE);
  c3 := UTL_I18N.GET_DEFAULT_CHARSET('French', 1, FALSE);
  DBMS_OUTPUT.PUT_LINE('GEN='||c1||' MAIL_WIN='||c2||' MAIL_NIX='||c3);
END;
/
```

UTL_I18N.GET_DEFAULT_ISO_CURRENCY

The function returns the default currency code (ISO 4217) for a given territory (database or session territory).

Input parameters:

- territory (VARCHAR2) – territory code (e.g., 'USA'). If NULL, default territory is used.

Output parameters:

- Returns a three-letter currency code (e.g., 'RUB', 'USD').

Typical usage scenarios:

- Determining the default currency for formatting monetary amounts in reports.
- Initializing parameters of financial applications depending on geographic context.
- Checking whether entered currency data matches expectations for the territory.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(10);
BEGIN
  v := UTL_I18N.GET_DEFAULT_ISO_CURRENCY('CHINA');
  DBMS_OUTPUT.PUT_LINE('CNY? ' || v);
END;
/
```

UTL_I18N.GET_DEFAULT_LINGUISTIC_SORT

The function returns the name of the default linguistic sort rule for the specified language context.

Input parameters:

- locale (VARCHAR2) – locale string (e.g., 'SPANISH').

Output parameters:

- Returns the sort parameter name (e.g., 'GENERIC_M').

Typical usage scenarios:

- Configuring sort rules (ORDER BY) for multilingual data in queries.
- Programmatically determining sort parameters for dynamically generated reports.
- Ensuring correct alphabetical grouping of data in different languages.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(50);
BEGIN
  v := UTL_I18N.GET_DEFAULT_LINGUISTIC_SORT('Japanese');
  DBMS_OUTPUT.PUT_LINE('SORT=' || v);
END;
/
```

UTL_I18N.GET_MAX_CHARACTER_SIZE

The function returns the maximum size in bytes required to store one character in the specified encoding.

Input parameters:

- charset (VARCHAR2) – character set name.

Output parameters:

- Returns the maximum character size in bytes (e.g., 1 for WE8MSWIN1252, 4 for AL32UTF8).

Typical usage scenarios:

- Calculating storage size for text fields considering multilingual content.
- Optimizing performance when processing large texts where byte operations matter.
- Checking external system limitations on string length in bytes.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  n PLS_INTEGER;
BEGIN
  n := UTL_I18N.GET_MAX_CHARACTER_SIZE('AL32UTF8');
  DBMS_OUTPUT.PUT_LINE('MAX=' || n);
END;
/
```

UTL_I18N.GET_TRANSLATION

The function returns the translation of a given text literal into the specified language, using Oracle's linguistic settings.

Input parameters:

- src_str (VARCHAR2) – source string to translate.
- src_lang (VARCHAR2) – source language of the string.
- dst_lang (VARCHAR2) – target language for translation.

Output parameters:

- Returns the translated string.

Typical usage scenarios:

- Dynamic localization of application messages and interfaces based on user language.
- Automatic translation of standard terms or names in reports.
- Supporting multilingual dictionaries within the database.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(200);
BEGIN
  v := UTL_I18N.GET_TRANSLATION('GERMAN_GERMANY', 'AMERICAN', 2);
  DBMS_OUTPUT.PUT_LINE('VAL=' || v);
END;
/
```

UTL_I18N.MAP_CHARSET

The function returns the canonical (official) Oracle character set name by its alias or name from another system.

Input parameters:

- charset (VARCHAR2) – encoding name or alias.

Output parameters:

- Returns the canonical encoding name in Oracle.

Typical usage scenarios:

- Normalizing encoding names received from external sources (e.g., from client applications, file headers).
- Ensuring compatibility when migrating data between different platforms or DBMS versions.
- Simplifying encoding handling logic by bringing them to a standard form.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(50);
BEGIN
  v := UTL_I18N.MAP_CHARSET('WE8DEC', 1, 1);
  DBMS_OUTPUT.PUT_LINE('MAIL=' || v);
END;
/
```

UTL_I18N.MAP_FROM_SHORT_LANGUAGE

The function converts a short (two-letter) language code (e.g., 'ru', 'en') to the full Oracle language name.

Input parameters:

- short_lang (VARCHAR2) – short language code (ISO 639-1).

Output parameters:

- Returns the full language name in Oracle (e.g., 'AMERICAN').

Typical usage scenarios:

- Converting language codes coming from web applications or international standards to Oracle's internal format for setting the session (NLS_LANGUAGE).
- Simplifying integration with external systems using short codes.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(50);
BEGIN
  v := UTL_I18N.MAP_FROM_SHORT_LANGUAGE('US');
  DBMS_OUTPUT.PUT_LINE('FULL='||v);
END;
/
```

UTL_I18N.MAP_LANGUAGE_FROM_ISO

The function converts an ISO standard language code (ISO 639-2/T or B) to an Oracle language name.

Input parameters:

- iso_language (VARCHAR2) – ISO 639-2 language code (three-letter).

Output parameters:

- Returns the Oracle language name.

Typical usage scenarios:

- Processing metadata from international sources (bibliographic systems, archives)

where ISO codes are used.

- Matching language settings when importing data from systems strictly following ISO standards.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(50);
BEGIN
  v := UTL_I18N.MAP_LANGUAGE_FROM_ISO('en_US');
  DBMS_OUTPUT.PUT_LINE('LANG=' || v);
END;
/
```

UTL_I18N.MAP_TO_SHORT_LANGUAGE

The function converts a full Oracle language name to a short (two-letter) language code (ISO 639-1).

Input parameters:

- ora_language (VARCHAR2) – language name in Oracle.

Output parameters:

- Returns the short language code (e.g., 'ru', 'en').

Typical usage scenarios:

- Preparing data for export to external systems, APIs, or web services expecting short language codes.
- Generating URLs, HTTP headers, or other metadata requiring standard language tags.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(10);
BEGIN
  v := UTL_I18N.MAP_TO_SHORT_LANGUAGE('american');
  DBMS_OUTPUT.PUT_LINE('SHORT='||v);
END;
/
```

UTL_I18N.TRANSLITERATE

The function converts text from one writing system to another (transliteration), e.g., from Cyrillic to Latin or vice versa.

Input parameters:

- str (VARCHAR2) – source string.
- src_script (VARCHAR2) – source writing system (e.g., 'Cyrillic').
- dst_script (VARCHAR2) – target writing system (e.g., 'Latin').

Output parameters:

- Returns the transliterated string.

Typical usage scenarios:

- Normalizing proper names (people's names, place names) for search or matching in international systems.
- Preparing data for systems that do not support certain alphabets (e.g., converting Cyrillic URLs to ASCII).
- Creating readable versions of text for users unfamiliar with the original alphabet.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(4000);
BEGIN
  v := UTL_I18N.TRANSLITERATE(
    '[ça va? naïve façade; coöperate; Straße; cœur; œuvre; æther; élève]',
    'latin_ascii_din91379');
  DBMS_OUTPUT.PUT_LINE('VAL=' || v);
END;
/
```

UTL_I18N.MAP_LOCALE_TO_ISO

The function converts an Oracle locale (combination of territory and language) to a corresponding locale string according to the ISO standard.

Input parameters:

- ora_locale (VARCHAR2) – locale in Oracle format.

Output parameters:

- Returns the locale string in ISO format.

Typical usage scenarios:

- Configuring internationalization (i18n) in applications that use standard ISO locales (e.g., Java, .NET).
- Exporting data with meta-information about language and territory in a format understandable to external systems.
- Creating configuration files or command-line parameters for utilities working with ISO locales.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v VARCHAR2(10);
BEGIN
  v := UTL_I18N.MAP_LOCALE_TO_ISO('American','America');
  DBMS_OUTPUT.PUT_LINE('ISO='||v);
END;
/
```

15.2.7. UTL_IDENT Package

UTL_IDENT.IS_ORACLE_SERVER

The function determines whether the code is executing on an Oracle Database server.

Input parameters:

- No parameters.

Output parameters:

- Returns TRUE if the code is executing on the DB server; FALSE otherwise.

Typical usage scenarios:

- Creating code that behaves differently when executed on the server vs. the client (e.g., in forms or reports).
- Conditional compilation or execution of server-specific operations (working with data, sending mail).
- Protecting against attempts to execute server code in an unintended environment.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
    v_val BOOLEAN;
BEGIN
    v_val := UTL_IDENT.IS_ORACLE_SERVER;
    IF NOT v_val THEN
        DBMS_OUTPUT.PUT_LINE('IS_ORACLE_SERVER() = FALSE');
    ELSE
        DBMS_OUTPUT.PUT_LINE('IS_ORACLE_SERVER() = TRUE');
    END IF;
END;
/
```

UTL_IDENT.IS_ORACLE_CLIENT

The function determines whether the code is executing in an Oracle client environment (e.g., Oracle Forms, Reports).

Input parameters:

- No parameters.

Output parameters:

- Returns TRUE if the code is executing in an Oracle client environment; FALSE otherwise.

Typical usage scenarios:

- Activating logic specific to the user interface (UI) only in client applications.
- Disabling resource-intensive server operations when executing on the client.
- Determining execution point for debugging distributed applications.

Query examples:


```
SET SERVEROUTPUT ON
DECLARE
    v_val BOOLEAN;
BEGIN
    v_val := UTL_IDENT.IS_ORACLE_CLIENT;
    IF NOT v_val THEN
        DBMS_OUTPUT.PUT_LINE('IS_ORACLE_CLIENT() = FALSE');
    ELSE
        DBMS_OUTPUT.PUT_LINE('IS_ORACLE_CLIENT() = TRUE');
    END IF;
END;
/
```

UTL_IDENT.IS_TIMESTEN

The function determines whether the code is executing in an Oracle TimesTen database (in-memory database).

Input parameters:

- No parameters.

Output parameters:

- Returns TRUE if the code is executing in TimesTen; FALSE otherwise.

Typical usage scenarios:

- Writing portable code that adapts its behavior to the features of an in-memory DBMS (query optimization, bypassing unsupported functions).
- Checking the environment before executing operations specific to TimesTen (e.g., working with the application cache).

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v_val BOOLEAN;
BEGIN
  v_val := UTL_IDENT.IS_TIMESTEN;
  IF NOT v_val THEN
    DBMS_OUTPUT.PUT_LINE('IS_TIMESTEN() = FALSE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('IS_TIMESTEN() = TRUE');
  END IF;
END;
/
```

UTL_IDENT.IS_ORACLE_FORMS

The function determines whether the code is executing in an Oracle Forms environment.

Input parameters:

- No parameters.

Output parameters:

- Returns TRUE if the code is executing in Oracle Forms; FALSE otherwise.

Typical usage scenarios:

- Activating logic related to Forms controls (triggers, validation) only in that environment.
- Preventing calls to server packages not intended for direct calls from Forms.
- Conditional debugging and logging in the context of a Forms application.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
    v_val BOOLEAN;
BEGIN
    v_val := UTL_IDENT.IS_ORACLE_FORMS;
    IF NOT v_val THEN
        DBMS_OUTPUT.PUT_LINE('IS_ORACLE_FORMS() = FALSE');
    ELSE
        DBMS_OUTPUT.PUT_LINE('IS_ORACLE_FORMS() = TRUE');
    END IF;
END;
/
```

15.2.8. UTL_INNADR Package

UTL_INADDR.GET_HOST_NAME;

The function returns the host name for a given IP address (DNS reverse lookup).

Input parameters:

- ip (VARCHAR2) – IP address in string format (e.g., '192.168.1.1').

Output parameters:

- Returns the host name corresponding to the IP address. If conversion is impossible, returns the original IP.

Typical usage scenarios:

- Determining the machine name of a connection source or request for audit and security purposes.
- Logging not only IP but also understandable server names in applications.
- Validating IP addresses by checking their registration in DNS.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
    v_name VARCHAR2(255);
BEGIN
    v_name := UTL_INADDR.GET_HOST_NAME;
    DBMS_OUTPUT.PUT_LINE('GET_HOST_NAME() = ' || v_name);
END;
/
```

UTL_INADDR.GET_HOST_ADDRESS;

The function returns the IP address for a given host name (DNS lookup).

Input parameters:

- hostname (VARCHAR2) – host name (e.g., 'oracle-server.company.com').

Output parameters:

- Returns the IP address corresponding to the host name. If conversion is impossible, an exception occurs.

Typical usage scenarios:

- Converting domain names to IP addresses for network operations (e.g., configuring ACL, checking availability).
- Determining the address of a target server before establishing a network connection from PL/SQL.
- Caching IP addresses to reduce delays associated with DNS.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
    v_addr VARCHAR2(255);
BEGIN
    v_addr := UTL_INADDR.GET_HOST_ADDRESS;
    DBMS_OUTPUT.PUT_LINE('GET_HOST_ADDRESS() = ' || v_addr);
END;
/
```

15.2.9. UTL_LMS Package

UTL_LMS.FORMAT_MESSAGE

The function formats a message by substituting arguments into a template containing placeholders (e.g., %1, %2). Analog of GET_MESSAGE with immediate substitution.

Input parameters:

- message_key (VARCHAR2) – message key in the Oracle message table or the template text itself.
- arg1, arg2, ... (VARCHAR2) – arguments for substitution into placeholders (up to 10 arguments).

Output parameters:

- Returns the formatted message.

Typical usage scenarios:

- Forming dynamic, localized error messages where specific values (object names, numbers, dates) are included in the text.
- Creating custom notifications and alerts with personalized content.
- Generating texts for reports or logs on the fly.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
    v_message VARCHAR2(4000);
BEGIN
    v_message := UTL_LMS.FORMAT_MESSAGE('Welcome, %s!', 'Alexey');
    DBMS_OUTPUT.PUT_LINE(v_message);
END;
/
```

UTL_LMS.GET_MESSAGE

The function retrieves message text by its key, usually from Oracle standard or user message tables, without automatic argument substitution.

Input parameters:

- `message_key` (VARCHAR2) – message key.
- `language` (VARCHAR2) – language for which to get the message (optional).

Output parameters:

- Returns the message text corresponding to the key and language.

Typical usage scenarios:

- Centralized management of application message texts (errors, hints, labels) through database tables.
- Implementing internationalization (i18n) mechanism where text depends on the user's session language.
- Retrieving standard Oracle system messages for subsequent processing or modification.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  s VARCHAR2(32767);
  rc INTEGER;
BEGIN
  rc := UTL_LMS.GET_MESSAGE(972, 'RDBMS', 'ORA', 'AMERICAN', s);
  DBMS_OUTPUT.PUT_LINE('rc=' || rc || ' msg=' || s);
END;
/
```

15.2.10. UTL_MATCH Package

UTL_MATCH.EDIT_DISTANCE

The function computes the edit distance (Levenshtein distance) between two strings. Returns the minimum number of insertion, deletion, and substitution operations required to transform one string into another.

Input parameters:

- `s1` (VARCHAR2) – first string.
- `s2` (VARCHAR2) – second string.

Output parameters:

- Returns the edit distance (0 for identical strings, >0 for different).

Typical usage scenarios:

- Finding approximate matches during data cleansing (e.g., correcting typos in names).
- Matching records from different sources where spelling discrepancies are possible.
- Assessing similarity of short text fields.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  test_var VARCHAR2(10) := 'A1';
  input_var VARCHAR2(200) := 'utl_match.edit_distance(''shackleford'',
''shackelford'')';
  descr_var VARCHAR2(100) := 'ED: Levenshtein classic';
  result_var NUMBER;
BEGIN
  result_var := UTL_MATCH.EDIT_DISTANCE('shackleford','shackelford');
  DBMS_OUTPUT.PUT_LINE(test_var||' | '||input_var||' | '||descr_var||' |
' ||result_var);
END;
/
```

UTL_MATCH.EDIT_DISTANCE_SIMILARITY

The function computes the percentage similarity of two strings based on edit distance. Returns a value from 0 (not similar at all) to 100 (identical).

Input parameters:

- s1 (VARCHAR2) – first string.
- s2 (VARCHAR2) – second string.

Output parameters:

- Returns the similarity percentage (0-100).

Typical usage scenarios:

- Ranking results of fuzzy search by degree of match.
- Setting threshold values for automatic decision making (e.g., "merge records if similarity > 90%").
- Visualizing the degree of string difference for the user.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  test_var  VARCHAR2(10)  := 'B1';
  input_var VARCHAR2(200) := 'utl_match.edit_distance_similarity(NULL, NULL)';
  descr_var VARCHAR2(100) := 'ED_SIM: both NULL → 100';
  result_var NUMBER;
BEGIN
  result_var := UTL_MATCH.EDIT_DISTANCE_SIMILARITY(NULL, NULL);
  DBMS_OUTPUT.PUT_LINE(test_var||' | '||input_var||' | '||descr_var||' | '||result_var);
END;
/
```

UTL_MATCH.JARO_WINKLER

The function computes the similarity of two strings using the Jaro-Winkler algorithm. The algorithm considers common characters and their positions, giving higher weight to matches at the beginning of strings. Returns a value from 0 to 1.

Input parameters:

- s1 (VARCHAR2) – first string.
- s2 (VARCHAR2) – second string.

Output parameters:

- Returns the Jaro-Winkler similarity coefficient.

Typical usage scenarios:

- Comparing people's names, company names, addresses where order and beginning of string are important.
- Using in data deduplication systems for customer or counterparty data.
- Combining with other metrics to improve accuracy of fuzzy matching.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  test_var  VARCHAR2(10)  := 'C1';
  input_var VARCHAR2(200) := 'utl_match.jaro_winkler(''hello'', ''world'')';
  descr_var VARCHAR2(100) := 'JW: float 0..1 (without rounding .8400001)';
  result_var BINARY_DOUBLE;
BEGIN
  result_var := UTL_MATCH.JARO_WINKLER('hello','world');
  DBMS_OUTPUT.PUT_LINE(test_var||' | '||input_var||' | '||descr_var||' | '||
NVL(TO_CHAR(result_var),'0'));
END;
/
```

UTL_MATCH.JARO_WINKLER_SIMILARITY

The function computes similarity using Jaro-Winkler and returns the result as an integer from 0 to 100 (percentage).

Input parameters:

- s1 (VARCHAR2) – first string.
- s2 (VARCHAR2) – second string.

Output parameters:

- Returns the Jaro-Winkler similarity percentage (0-100).

Typical usage scenarios:

- Same scenarios as for JARO_WINKLER, but with output in a more familiar percentage format for reports or threshold checks.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  test_var VARCHAR2(10) := 'D1';
  input_var VARCHAR2(200) := 'utl_match.jaro_winkler_similarity(''shackleford'',
''shackelford'');
  descr_var VARCHAR2(100) := 'JW_SIM: percentages, prefix boost';
  result_var NUMBER;
BEGIN
  result_var := UTL_MATCH.JARO_WINKLER_SIMILARITY('shackleford','shackelford');
  DBMS_OUTPUT.PUT_LINE(test_var||' | '||input_var||' | '||descr_var||' |
' ||result_var);
END;
/
```

15.2.11. UTL_RAW Package

UTL_RAW.BIT_AND

The function performs bitwise "AND" operation on two RAW values.

Input parameters:

- r1 (RAW) – first value.
- r2 (RAW) – second value.

Output parameters:

- Returns the result of bitwise AND.

Typical usage scenarios:

- Applying bit masks to binary data (e.g., extracting specific flags from a status byte).
- Low-level processing of network packets or protocol headers.
- Implementing cryptographic algorithms or hash operations.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(2000);
BEGIN
  v := UTL_RAW.BIT_AND(HEXTORAW('F0F00F0F'), HEXTORAW('0FF00FF0'));
  DBMS_OUTPUT.PUT_LINE('BIT_AND = ' || RAWTOHEX(v));
END;
/
```

UTL_RAW.BIT_OR

The function performs bitwise "OR" operation on two RAW values.

Input parameters:

- r1 (RAW) – first value.
- r2 (RAW) – second value.

Output parameters:

- Returns the result of bitwise OR.

Typical usage scenarios:

- Setting specific bits in binary data (e.g., enabling flags).
- Combining masks or bit fields.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(2000);
BEGIN
  v := UTL_RAW.BIT_OR(HEXTORAW('F0F00F0F'), HEXTORAW('0FF00FF0'));
  DBMS_OUTPUT.PUT_LINE('BIT_OR = ' || RAWTOHEX(v));
END;
/
```

UTL_RAW.BIT_XOR

The function performs bitwise "exclusive OR" (XOR) operation on two RAW values.

Input parameters:

- r1 (RAW) – first value.
- r2 (RAW) – second value.

Output parameters:

- Returns the result of bitwise XOR.

Typical usage scenarios:

- Simple encryption/decryption of data using a key (Vernam cipher).
- Parity checking, checksum calculation.
- Comparing data: XOR result will be zero for identical values.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(2000);
BEGIN
  v := UTL_RAW.BIT_XOR(HEXTORAW('F0F00F0F'), HEXTORAW('0FF00FF0'));
  DBMS_OUTPUT.PUT_LINE('BIT_XOR = ' || RAWTOHEX(v));
END;
/
```

UTL_RAW.BIT_COMPLEMENT

The function performs bitwise "NOT" (complement) operation on a RAW value, inverting all bits.

Input parameters:

- r (RAW) – source value.

Output parameters:

- Returns the inverted value.

Typical usage scenarios:

- Inverting bit masks.
- Transforming data within specific algorithms or protocols.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  v RAW(2000);
BEGIN
  v := UTL_RAW.BIT_COMPLEMENT(HEXTORAW('00FF'));
  DBMS_OUTPUT.PUT_LINE('BIT_COMPLEMENT = ' || RAWTOHEX(v));
END;
/
```

UTL_RAW.CAST_FROM_BINARY_DOUBLE

The function converts a value of type BINARY_DOUBLE to RAW.

Input parameters:

- b (BINARY_DOUBLE) – double-precision floating-point value.

Output parameters:

- Returns the binary representation of the number.

Typical usage scenarios:

- Preparing floating-point numeric data for cryptographic operations that require RAW input.
- Saving or transmitting numbers in exact binary format without loss.
- Low-level work with data where internal number representation is important.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.CAST_FROM_BINARY_DOUBLE(123.25, 1);
  DBMS_OUTPUT.PUT_LINE('CAST_FROM_BINARY_DOUBLE (BE) = ' || RAWTOHEX(r));
END;
/
```

UTL_RAW.CAST_TO_BINARY_DOUBLE

The function converts a value of type RAW to BINARY_DOUBLE.

Input parameters:

- r (RAW) – binary data representing a BINARY_DOUBLE number.

Output parameters:

- Returns the restored floating-point number.

Typical usage scenarios:

- Restoring numbers after cryptographic operations or transmission in RAW format.
- Reading binary data received from external systems where numbers are stored in standard IEEE format.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
  d BINARY_DOUBLE;
BEGIN
  r := UTL_RAW.CAST_FROM_BINARY_DOUBLE(123.25, 1);
  d := UTL_RAW.CAST_TO_BINARY_DOUBLE(r, 1);
  DBMS_OUTPUT.PUT_LINE('CAST_TO_BINARY_DOUBLE (round-trip) = ' || d);
END;
/
```

UTL_RAW.CAST_FROM_BINARY_FLOAT

The function converts a value of type BINARY_FLOAT to RAW.

Input parameters:

- b (BINARY_FLOAT) – single-precision floating-point value.

Output parameters:

- Returns the binary representation of the number.

Typical usage scenarios:

- Similar to CAST_FROM_BINARY_DOUBLE, but for single-precision numbers.

- Optimizing memory/bandwidth usage when working with large arrays of numbers.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.CAST_FROM_BINARY_FLOAT(3.5, 2);
  DBMS_OUTPUT.PUT_LINE('CAST_FROM_BINARY_FLOAT (LE) = ' || RAWTOHEX(r));
END;
/
```

UTL_RAW.CAST_TO_BINARY_FLOAT

The function converts a value of type RAW to BINARY_FLOAT.

Input parameters:

- r (RAW) – binary data representing a BINARY_FLOAT number.

Output parameters:

- Returns the restored floating-point number.

Typical usage scenarios:

- Restoring single-precision numbers from their binary representation.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
  f BINARY_FLOAT;
BEGIN
  r := UTL_RAW.CAST_FROM_BINARY_FLOAT(3.5, 2);
  f := UTL_RAW.CAST_TO_BINARY_FLOAT(r, 2);
  DBMS_OUTPUT.PUT_LINE('CAST_TO_BINARY_FLOAT (round-trip) = ' || f);
END;
/
```

UTL_RAW.CAST_FROM_BINARY_INTEGER

The function converts a value of type `BINARY_INTEGER` or `PLS_INTEGER` to `RAW`.

Input parameters:

- `b` (`BINARY_INTEGER`) – integer value.

Output parameters:

- Returns the binary representation of the number.

Typical usage scenarios:

- Converting integers for operations with `RAW` data (hashing, encryption).
- Preparing parameters for low-level APIs or custom aggregate functions.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.CAST_FROM_BINARY_INTEGER(-42, 1);
  DBMS_OUTPUT.PUT_LINE('CAST_FROM_BINARY_INTEGER (BE) = ' || RAWTOHEX(r));
END;
/
```

UTL_RAW.CAST_TO_BINARY_INTEGER

The function converts a value of type `RAW` to `BINARY_INTEGER`.

Input parameters:

- `r` (`RAW`) – binary data representing an integer.

Output parameters:

- Returns the restored integer.

Typical usage scenarios:

- Restoring integer parameters after processing in `RAW` format.
- Extracting numeric values from binary data structures.

Query examples:


```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
  i BINARY_INTEGER;
BEGIN
  r := UTL_RAW.CAST_FROM_BINARY_INTEGER(-42, 1);
  i := UTL_RAW.CAST_TO_BINARY_INTEGER(r, 1);
  DBMS_OUTPUT.PUT_LINE('CAST_TO_BINARY_INTEGER (round-trip) = ' || i);
END;
/
```

UTL_RAW.CAST_TO_VARCHAR2

The function converts a value of type RAW to VARCHAR2. Each byte of RAW is interpreted as a character in the database encoding (most often a single-byte encoding for RAW $\Leftarrow$ 2000 bytes).

Input parameters:

- r (RAW) – binary data.

Output parameters:

- Returns a string obtained by direct mapping of bytes to characters.

Typical usage scenarios:

- Quick conversion of RAW data guaranteed to contain only single-byte characters (e.g., hexadecimal strings, Base64, ASCII codes).
- Debugging and logging RAW variable contents in readable form (if content is text).

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  s VARCHAR2(2000);
BEGIN
  s := UTL_RAW.CAST_TO_VARCHAR2(HEXTORAW('48656C6C6F2C2052415721')); -- "Hello, RAW!"
  DBMS_OUTPUT.PUT_LINE('CAST_TO_VARCHAR2 = ' || s);
END;
/
```

UTL_RAW.CAST_TO_RAW

The function converts a VARCHAR2 string to RAW type. Each character of the string is converted to a byte based on the database encoding. Important: Suitable only for strings containing characters from a single-byte set.

Input parameters:

- c (VARCHAR2) – source string.

Output parameters:

- Returns binary data.

Typical usage scenarios:

- Converting text strings containing only Latin-1 or ASCII characters to RAW for further processing.
- Creating test or constant RAW values from string literals.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.CAST_TO_RAW('Hello RAW');
  DBMS_OUTPUT.PUT_LINE('CAST_TO_RAW = ' || RAWTOHEX(r));
END;
/
```

UTL_RAW.COMPARE

The function compares two RAW values. Returns 0 if they are identical. If they differ, returns the position of the first differing byte (starting from 1). If one value is a prefix of the other, returns the length of the shorter value + 1.

Input parameters:

- r1 (RAW) – first value.
- r2 (RAW) – second value.

Output parameters:

- Returns the comparison result (0 – equal, >0 – position of first difference).

Typical usage scenarios:

- Byte-by-byte comparison of binary data (hashes, encrypted blocks, serialized objects).
- Determining whether one value is the beginning of another (prefix check).

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  n NUMBER;
BEGIN
  n := UTL_RAW.COMPARE(HEXTORAW('AA00'), HEXTORAW('AA01'));
  DBMS_OUTPUT.PUT_LINE('COMPARE = ' || n);
END;
/
```

UTL_RAW.CONCAT

The function concatenates multiple RAW values into one.

Input parameters:

- r1 (RAW) – first value.
- r2 (RAW) – second value.
- r3 ... rN (RAW) – additional values (up to 12).

Output parameters:

- Returns the result of concatenating all arguments.

Typical usage scenarios:

- Assembling binary packets or messages from separate parts (header + body + signature).
- Building keys or input data for cryptographic functions by combining several elements.
- Dynamically forming binary content for attachments or BLOBs.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(32767);
BEGIN
  r := UTL_RAW.CONCAT(HEXTORAW('AA'), HEXTORAW('BBCC'), NULL);
  DBMS_OUTPUT.PUT_LINE('CONCAT = ' || RAWTOHEX(r));
END;
/
```

UTL_RAW.COPIES

The function returns a RAW value consisting of the specified number of repetitions of the source RAW value.

Input parameters:

- r (RAW) – pattern to repeat.
- n (PLS_INTEGER) – number of repetitions.

Output parameters:

- Returns concatenated copies of the source value.

Typical usage scenarios:

- Creating test data of a given size (e.g., placeholders, patterns).
- Generating constant masks or buffers for initialization.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(32767);
BEGIN
  r := UTL_RAW.COPIES(HEXTORAW('DEAD'), 3);
  DBMS_OUTPUT.PUT_LINE('COPIES = ' || RAWTOHEX(r)); -- DEADDEADDEAD
END;
/
```

UTL_RAW.LENGTH

The function returns the length of a RAW value in bytes.

Input parameters:

- r (RAW) – source value.

Output parameters:

- Returns the length in bytes.

Typical usage scenarios:

- Checking the size of binary data before processing, transmitting, or storing it.
- Validating input parameters (e.g., hash length must be 32 bytes).
- Allocating memory or buffers based on data size.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  n NUMBER;
BEGIN
  n := UTL_RAW.LENGTH(HEXTORAW('0011223344'));
  DBMS_OUTPUT.PUT_LINE('LENGTH = ' || n); -- 5
END;
/
```

UTL_RAW.OVERLAY

The function replaces part of the target RAW value, starting from the specified position, with the specified replacement RAW value.

Input parameters:

- target (RAW) – target value to be modified.
- overlay (RAW) – replacement value.
- position (PLS_INTEGER) – position in the target value where replacement starts (default 1).
- length (PLS_INTEGER) – number of bytes in the target value to replace (default length of replacement value).

Output parameters:

- Returns the modified value.

Typical usage scenarios:

- Modifying binary data structures (e.g., updating a field in a file or packet header).
- "Patching" or masking certain areas in binary data.
- Inserting or replacing blocks in a data stream.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.OVERLAY(HEXTORAW('DEAD'), HEXTORAW('0011223344'), 4, 3);
  DBMS_OUTPUT.PUT_LINE('OVERLAY = ' || RAWTOHEX(r)); -- 0011DEAD44
END;
/
```

UTL_RAW.REVERSE

The function returns a RAW value where the byte order is reversed from the original.

Input parameters:

- r (RAW) – source value.

Output parameters:

- Returns the value with reversed byte order.

Typical usage scenarios:

- Converting byte order (endianness) between systems (little-endian $\leftrightarrow$ big-endian).
- Processing data written in reverse order.
- Specific algorithms requiring byte sequence reversal.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.REVERSE(HEXTORAW('0102F3'));
  DBMS_OUTPUT.PUT_LINE('REVERSE = ' || RAWTOHEX(r)); -- F30201
END;
/
```

UTL_RAW.SUBSTR

The function extracts a substring (subsequence of bytes) from a RAW value.

Input parameters:

- r (RAW) – source value.
- pos (PLS_INTEGER) – starting position for extraction (default 1).
- len (PLS_INTEGER) – number of bytes to extract. If not specified or exceeds available amount, all bytes to the end are extracted.

Output parameters:

- Returns the extracted part.

Typical usage scenarios:

- Extracting individual fields from a binary structure (e.g., first 4 bytes as header).
- Parsing incoming binary messages or files.
- Getting part of a hash value or signature.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.SUBSTR(HEXTORAW('001122334455'), 3, 2);
  DBMS_OUTPUT.PUT_LINE('SUBSTR = ' || RAWTOHEX(r)); -- 1122
END;
/
```

UTL_RAW.TRANSLATE

The function transforms a RAW value, replacing each byte found in the "from_set" string with the corresponding byte from the "to_set" string. Analog of TRANSLATE for strings, but for bytes.

Input parameters:

- r (RAW) – source value.
- from_set (RAW) – set of bytes to search for.
- to_set (RAW) – set of bytes to replace with.

Output parameters:

- Returns the transformed value.

Typical usage scenarios:

- Simple replacement or masking of specific bytes in binary data (e.g., removing null bytes).
- Applying simple substitution ciphers at the byte level.
- Normalizing data (bringing different codes of the same character to a unified form).

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.TRANSLATE(HEXTORAW('00112233'),
                        HEXTORAW('001133'),
                        HEXTORAW('FF22'));
  DBMS_OUTPUT.PUT_LINE('TRANSLATE = ' || RAWTOHEX(r)); -- FF22
END;
/
```

UTL_RAW.TRANSLITERATE

The function converts a RAW value from one character set to another. This is a byte-by-byte operation that can change the length of the result if encodings have different

character sizes (e.g., single-byte to UTF-8).

Input parameters:

- src (RAW) – source binary data.
- src_charset (VARCHAR2) – source encoding.
- dst_charset (VARCHAR2) – target encoding.

Output parameters:

- RAW – converted data in the target encoding.

Typical usage scenarios:

- Converting file or BLOB content between various encodings at the byte level.
- Preparing text data for systems requiring a specific encoding.
- Fixing "garbage characters" problems when data was saved in the wrong encoding.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(2000);
BEGIN
  r := UTL_RAW.TRANSLITERATE(HEXTORAW('00112233'),
                             HEXTORAW('FF22'),
                             HEXTORAW('001133'),
                             HEXTORAW('99'));
  DBMS_OUTPUT.PUT_LINE('TRANSLITERATE = ' || RAWTOHEX(r)); -- FF229933
END;
/
```

UTL_RAW.XRANGE

The function returns a RAW value representing a sequence of bytes from the start byte to the end byte inclusive.

Input parameters:

- start_byte (RAW) – start byte (one byte, e.g., HEXTORAW('00')).
- end_byte (RAW) – end byte (one byte, e.g., HEXTORAW('FF')).

Output parameters:

- Returns a sequence of bytes from start_byte to end_byte.

Typical usage scenarios:

- Generating test data containing all possible byte values.
- Creating lookup tables or masks for processing algorithms.
- Initializing buffers with sequential values.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  r RAW(32767);
BEGIN
  r := UTL_RAW.XRANGE(HEXTORAW('FA'), HEXTORAW('02'));
  DBMS_OUTPUT.PUT_LINE('XRange = ' || RAWTOHEX(r)); -- FAFBFCFDFEFF000102
END;
/
```

15.2.12. UTL_SMTP Package

UTL_SMTP.OPEN_CONNECTION

The function opens a network connection to the specified SMTP server for subsequent email sending. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- host (VARCHAR2) – host name or IP address of the SMTP server.
- port (BINARY_INTEGER) – port number of the SMTP server (default 25).
- c (PL/SQL RECORD) – output variable into which the created connection object (SMTP connection) will be placed.
- hosts (VARCHAR2) – list of hosts for connecting through one of several addresses.
- secure_connection (BOOLEAN) – flag indicating the need to establish a secure (SSL) connection.

Output parameters:

- Returns a connection object of type SMTP_CONNECTION, used in other procedures of the UTL_SMTP package.

Typical usage scenarios:

- Setting a non-standard port for connecting to the SMTP server.
- Organizing email sending through a server requiring authentication (after opening the connection).
- Configuring a secure SSL connection with the mail server.
- Programmatically implementing a complex sending scenario not covered by UTL_MAIL.SEND.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Connecting');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  DBMS_OUTPUT.PUT_LINE('Connected');
END;
/
```

UTL_SMTP.CLOSE_CONNECTION

The procedure closes an active SMTP connection established earlier with the OPEN_CONNECTION function. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – object of type SMTP_CONNECTION representing the current connection.

Output parameters:

- None.

Typical usage scenarios:

- Ending work with the SMTP server after sending an email.
- Releasing resources associated with an active connection.
- Correctly closing the connection at the end of an SMTP session.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('OPEN_CONNECTION');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  DBMS_OUTPUT.PUT_LINE('CLOSE_CONNECTION');
  UTL_SMTP.CLOSE_CONNECTION(c);
  DBMS_OUTPUT.PUT_LINE('Success: connection opened and closed...');
END;
/
```

UTL_SMTP.QUIT

The procedure sends the SMTP QUIT command to the server to end the session and close the connection. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.

Output parameters:

- None.

Typical usage scenarios:

- Ending an SMTP session according to protocol before closing the connection.
- Correctly notifying the server of client termination.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('OPEN_CONNECTION');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  DBMS_OUTPUT.PUT_LINE('Quit');
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Successfully closed via Quit');
END;
/
```

UTL_SMTP.NOOP

The procedure sends the NOOP command to the server to check connection activity without performing other actions. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.

Output parameters:

- None.

Typical usage scenarios:

- Checking that the connection with the server remains active.
- Maintaining the session during long sending of large emails.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('OPEN_CONNECTION');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  DBMS_OUTPUT.PUT_LINE('NOOP');
  UTL_SMTP.NOOP(c);
  DBMS_OUTPUT.PUT_LINE('NOOP executed successfully');
  DBMS_OUTPUT.PUT_LINE('Quit');
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Successfully closed via Quit');
END;
/
```

UTL_SMTP.HELO

The procedure sends the SMTP HELO command to the server to start dialogue and identify the client by host name. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.
- domain (VARCHAR2) – client domain name transmitted to the server.

Output parameters:

- None.

Typical usage scenarios:

- Initializing a standard SMTP session without extensions.
- Establishing basic interaction between client and server.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('OPEN_CONNECTION');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  DBMS_OUTPUT.PUT_LINE('HELO');
  UTL_SMTP.HELO(c, 'example.com'); -- Basic greeting
  DBMS_OUTPUT.PUT_LINE('QUIT');
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: performed HELO and QUIT on server 127.0.0.1:2525');
END;
/
```

UTL_SMTP.EHLO

The procedure sends the SMTP EHLO command to the server, identical to HELO, but with support for ESMTP protocol extensions. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.
- domain (VARCHAR2) – client domain name.

Output parameters:

- None.

Typical usage scenarios:

- Using SMTP extensions such as authorization or large attachment transmission.
- Initializing secure and extended SMTP sessions.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('OPEN_CONNECTION');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  DBMS_OUTPUT.PUT_LINE('EHLO');
  UTL_SMTP.EHLO(c, 'example.com'); -- Extended greeting
  DBMS_OUTPUT.PUT_LINE('QUIT');
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: performed EHLO and QUIT on server 127.0.0.1');
END;
/
```

UTL_SMTP.MAIL

The procedure sends the SMTP MAIL FROM command to specify the sender's email address. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.
- sender (VARCHAR2) – sender's email address.
- parameters (VARCHAR2) – additional parameters (e.g., SIZE).

Output parameters:

- None.

Typical usage scenarios:

- Defining the sender of an email in an SMTP session.
- Configuring special sending parameters at the protocol level.

Query examples:


```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== MAIL + RCPT + RSET TEST ===');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  UTL_SMTP.EHLO(c, 'example.com');
  -- Set sender and recipient
  DBMS_OUTPUT.PUT_LINE('MAIL FROM: test@example.com');
  UTL_SMTP.MAIL(c, 'test@example.com', NULL); -- Third parameter = NULL
  DBMS_OUTPUT.PUT_LINE('RCPT TO: user@example.com');
  UTL_SMTP.RCPT(c, 'user@example.com');
  -- Reset state (cancel previous MAIL/RCPT)
  DBMS_OUTPUT.PUT_LINE('RSET - reset state');
  UTL_SMTP.RSET(c);
  -- After RSET, new parameters can be set
  DBMS_OUTPUT.PUT_LINE('MAIL FROM: new@example.com');
  UTL_SMTP.MAIL(c, 'new@example.com', NULL); -- Third parameter = NULL
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: MAIL+RCPT+RSET checked');
END;
/
```

UTL_SMTP.RCPT

The procedure sends the SMTP RCPT TO command to specify the recipient of the message. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.
- recipient (VARCHAR2) – recipient's email address.
- parameters(VARCHAR2) - protocol extension (SMTP extensions)

Output parameters:

- None.

Typical usage scenarios:

- Adding one or multiple recipients to an email.
- Specifying addressees for mass mailing.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== MAIL + RCPT + RSET TEST ===');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  UTL_SMTP.EHLO(c, 'example.com');
  -- Set sender and recipient
  DBMS_OUTPUT.PUT_LINE('MAIL FROM: test@example.com');
  UTL_SMTP.MAIL(c, 'test@example.com', NULL); -- Third parameter = NULL
  DBMS_OUTPUT.PUT_LINE('RCPT TO: user@example.com');
  UTL_SMTP.RCPT(c, 'user@example.com');
  -- Reset state (cancel previous MAIL/RCPT)
  DBMS_OUTPUT.PUT_LINE('RSET - reset state');
  UTL_SMTP.RSET(c);
  -- After RSET, new parameters can be set
  DBMS_OUTPUT.PUT_LINE('MAIL FROM: new@example.com');
  UTL_SMTP.MAIL(c, 'new@example.com', NULL); -- Third parameter = NULL
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: MAIL+RCPT+RSET checked');
END;
/
```

UTL_SMTP.RSET

The procedure sends the SMTP RSET command to the server to reset the current session state without closing the connection. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.

Output parameters:

- None.

Typical usage scenarios:

- Clearing state after a sending error.
- Preparing the connection for a new email without reopening the connection.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== MAIL + RCPT + RSET TEST ===');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  UTL_SMTP.EHLO(c, 'example.com');
  -- Set sender and recipient
  DBMS_OUTPUT.PUT_LINE('MAIL FROM: test@example.com');
  UTL_SMTP.MAIL(c, 'test@example.com', NULL); -- Third parameter = NULL
  DBMS_OUTPUT.PUT_LINE('RCPT TO: user@example.com');
  UTL_SMTP.RCPT(c, 'user@example.com');
  -- Reset state (cancel previous MAIL/RCPT)
  DBMS_OUTPUT.PUT_LINE('RSET - reset state');
  UTL_SMTP.RSET(c);
  -- After RSET, new parameters can be set
  DBMS_OUTPUT.PUT_LINE('MAIL FROM: new@example.com');
  UTL_SMTP.MAIL(c, 'new@example.com', NULL); -- Third parameter = NULL
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: MAIL+RCPT+RSET checked');
END;
/
```

UTL_SMTP.DATA

The procedure sends the SMTP DATA command, switching the connection to email content transmission mode. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.
- data (VARCHAR2) – email text (including headers and body).

Output parameters:

- None.

Typical usage scenarios:

- Sending a ready email in one call.

- Completing email formation within an SMTP session.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== SIMPLE DATA TEST ===');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  UTL_SMTP.EHLO(c, 'example.com');
  UTL_SMTP.MAIL(c, 'sender@example.com', NULL);
  UTL_SMTP.RCPT(c, 'receiver@example.com');
  -- Simplest DATA variant
  DBMS_OUTPUT.PUT_LINE('DATA ONE-SHOT');
  UTL_SMTP.DATA(c, 'Test email body. ');
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: simple DATA checked');
END;
/
```

UTL_SMTP.OPEN_DATA

The procedure initiates a block of email data transmission after the DATA command, opening a stream for writing content. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.

Output parameters:

- Returns a descriptor of the opened stream for writing email data.

Typical usage scenarios:

- Sending large messages in parts.
- Sequentially writing headers and email body to the stream.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== OPEN_DATA + WRITE_DATA + CLOSE_DATA TEST ===');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  UTL_SMTP.EHLO(c, 'example.com');
  UTL_SMTP.MAIL(c, 'sender@example.com', NULL);
  UTL_SMTP.RCPT(c, 'receiver@example.com');
  -- Start data transmission
  DBMS_OUTPUT.PUT_LINE('OPEN_DATA');
  UTL_SMTP.OPEN_DATA(c);
  -- Stage-by-stage data writing
  DBMS_OUTPUT.PUT_LINE('WRITE_DATA');
  UTL_SMTP.WRITE_DATA(c, 'Subject: Test Stream Data');
  UTL_SMTP.WRITE_DATA(c, 'From: sender@example.com');
  UTL_SMTP.WRITE_DATA(c, 'To: receiver@example.com');
  UTL_SMTP.WRITE_DATA(c, ''); -- Empty line
  UTL_SMTP.WRITE_DATA(c, 'This is line 1 of the email body.');
  UTL_SMTP.WRITE_DATA(c, 'This is line 2 of the email body.');
  UTL_SMTP.WRITE_DATA(c, 'This is line 3 of the email body.');
  -- Finish transmission
  DBMS_OUTPUT.PUT_LINE('CLOSE_DATA');
  UTL_SMTP.CLOSE_DATA(c);
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: stage-by-stage sending checked');
END;
/
```

UTL_SMTP.WRITE_DATA

The procedure writes part of email data to the opened stream created by the OPEN_DATA call. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.
- data (VARCHAR2) – string with part of email content.

Output parameters:

- None.

Typical usage scenarios:

- Stage-by-stage sending of large emails or binary attachments.
- Managing structure and formatting of email at the stream level.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== OPEN_DATA + WRITE_DATA + CLOSE_DATA TEST ===');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  UTL_SMTP.EHLO(c, 'example.com');
  UTL_SMTP.MAIL(c, 'sender@example.com', NULL);
  UTL_SMTP.RCPT(c, 'receiver@example.com');
  -- Start data transmission
  DBMS_OUTPUT.PUT_LINE('OPEN_DATA');
  UTL_SMTP.OPEN_DATA(c);
  -- Stage-by-stage data writing
  DBMS_OUTPUT.PUT_LINE('WRITE_DATA');
  UTL_SMTP.WRITE_DATA(c, 'Subject: Test Stream Data');
  UTL_SMTP.WRITE_DATA(c, 'From: sender@example.com');
  UTL_SMTP.WRITE_DATA(c, 'To: receiver@example.com');
  UTL_SMTP.WRITE_DATA(c, ''); -- Empty line
  UTL_SMTP.WRITE_DATA(c, 'This is line 1 of the email body. ');
  UTL_SMTP.WRITE_DATA(c, 'This is line 2 of the email body. ');
  UTL_SMTP.WRITE_DATA(c, 'This is line 3 of the email body. ');
  -- Finish transmission
  DBMS_OUTPUT.PUT_LINE('CLOSE_DATA');
  UTL_SMTP.CLOSE_DATA(c);
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: stage-by-stage sending checked');
END;
/
```

UTL_SMTP.CLOSE_DATA

The procedure finishes email data transmission, closing the stream opened by the OPEN_DATA function. Attention! This procedure requires configuring the SMTP server parameter in the postgresql.conf file. Read more in chapter 4.5.2. Database Directory.

Input parameters:

- c (PL/SQL RECORD) – active SMTP connection.

Output parameters:

- None.

Typical usage scenarios:

- Finishing writing email body and transmitting it to the server.
- Confirming end of data in SMTP session before sending QUIT command.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  c UTL_SMTP.CONNECTION;
BEGIN
  DBMS_OUTPUT.PUT_LINE('=== OPEN_DATA + WRITE_DATA + CLOSE_DATA TEST ===');
  c := UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 2525);
  UTL_SMTP.EHLO(c, 'example.com');
  UTL_SMTP.MAIL(c, 'sender@example.com', NULL);
  UTL_SMTP.RCPT(c, 'receiver@example.com');
  -- Start data transmission
  DBMS_OUTPUT.PUT_LINE('OPEN_DATA');
  UTL_SMTP.OPEN_DATA(c);
  -- Stage-by-stage data writing
  DBMS_OUTPUT.PUT_LINE('WRITE_DATA');
  UTL_SMTP.WRITE_DATA(c, 'Subject: Test Stream Data');
  UTL_SMTP.WRITE_DATA(c, 'From: sender@example.com');
  UTL_SMTP.WRITE_DATA(c, 'To: receiver@example.com');
  UTL_SMTP.WRITE_DATA(c, ''); -- Empty line
  UTL_SMTP.WRITE_DATA(c, 'This is line 1 of the email body. ');
  UTL_SMTP.WRITE_DATA(c, 'This is line 2 of the email body. ');
  UTL_SMTP.WRITE_DATA(c, 'This is line 3 of the email body. ');
  -- Finish transmission
  DBMS_OUTPUT.PUT_LINE('CLOSE_DATA');
  UTL_SMTP.CLOSE_DATA(c);
  UTL_SMTP.QUIT(c);
  DBMS_OUTPUT.PUT_LINE('Success: stage-by-stage sending checked');
END;
/
```

15.2.13. UTL_TCP Package

UTL_TCP.OPEN_CONNECTION

The function opens a TCP connection to a remote server and returns a connection descriptor for use in other functions of the package.

Input parameters:

- remote_host (VARCHAR2) – host name or IP address of the remote server.
- remote_port (PLS_INTEGER) – port number on the remote server.
- local_host (VARCHAR2) – host name or IP address of the local interface for outgoing connection (optional).
- local_port (PLS_INTEGER) – local port number (optional, usually 0).
- charset (VARCHAR2) – character set for text exchange (default taken from the session).
- newline (CHAR) – end-of-line character(s) (CRLF by default).
- tx_timeout (PLS_INTEGER) – send data timeout in seconds.
- rx_timeout (PLS_INTEGER) – receive data timeout in seconds.

Output parameters:

- connection (PLS_INTEGER) – descriptor of the opened connection.

Typical usage scenarios:

- Interacting with web services (HTTP/HTTPS) over TCP protocols.
- Exchanging data with sockets of other applications (Java, .NET, Python).
- Integration with third-party systems providing TCP interfaces.
- Sending data to remote monitoring or logging servers.

Query examples:


```
SET SERVEROUTPUT ON
DECLARE
  h UTL_TCP.CONNECTION;
BEGIN
  h := UTL_TCP.OPEN_CONNECTION('10.20.66.82', 5678);
  DBMS_OUTPUT.PUT_LINE('OPEN_CONNECTION: OK');
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.CLOSE_CONNECTION

The procedure closes the specified TCP connection.

Input parameters:

- c (PLS_INTEGER) – connection descriptor returned by OPEN_CONNECTION.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Correctly terminating a network connection after finishing data exchange.
- Releasing network resources on both server and client sides.
- Handling errors for guaranteed connection closure.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h UTL_TCP.CONNECTION;
BEGIN
  h := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  d := UTL_TCP.CLOSE_CONNECTION_P(h);
  DBMS_OUTPUT.PUT_LINE('CLOSE_CONNECTION: ok');
END;
/
```

UTL_TCP.WRITE_TEXT

The procedure sends text data (VARCHAR2) through an open TCP connection.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.
- data (VARCHAR2) – text to send.
- len (PLS_INTEGER) – number of characters to send. If NULL, sends the entire string.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Sending text queries to an HTTP server or other text protocols (SMTP, FTP commands).
- Transmitting structured data in XML or JSON format.
- Sending logs or notifications to a remote server.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h    UTL_TCP.CONNECTION;
  d    INTEGER;
  sent PLS_INTEGER;
BEGIN
  h    := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  sent := UTL_TCP.WRITE_TEXT(h, 'buffer', NULL);
  d    := UTL_TCP.FLUSH(h);
  DBMS_OUTPUT.PUT_LINE('FLUSH: ok, sent=' || sent);
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.GET_TEXT

The function reads text data from a TCP connection.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.
- len (PLS_INTEGER) – maximum number of characters to read. If NULL, reads all available data until timeout.

Output parameters:

- Returns the read text data.

Typical usage scenarios:

- Receiving text responses from a web server (HTML, JSON, XML).
- Reading string data from another application's socket.
- Receiving status messages from a remote service.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h    UTL_TCP.CONNECTION;
  txt  VARCHAR2(4000);
  s    PLS_INTEGER;
BEGIN
  h    := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  s    := UTL_TCP.WRITE_TEXT(h, 'xyz', NULL);
  txt  := UTL_TCP.GET_TEXT(h, 3);
  DBMS_OUTPUT.PUT_LINE('GET_TEXT="' || txt || '"');
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.FLUSH

The procedure forcibly sends all buffered data to the socket.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Guaranteed sending of data before expecting a response from the server.
- Optimizing performance when sending many small messages (sending as one packet).

- Working in non-buffering mode (if immediate sending is required).

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h    UTL_TCP.CONNECTION;
  d    INTEGER;
  sent PLS_INTEGER;
BEGIN
  h    := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  sent := UTL_TCP.WRITE_TEXT(h, 'buffer', NULL);
  d    := UTL_TCP.FLUSH(h);
  DBMS_OUTPUT.PUT_LINE('FLUSH: ok, sent=' || sent);
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.AVAILABLE

The function returns the number of bytes available for reading from the socket without blocking.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.

Output parameters:

- PLS_INTEGER – number of available bytes.

Typical usage scenarios:

- Checking for data before attempting to read to avoid blocking.
- Implementing non-blocking reading or polling loop.
- Determining the size of an expected response from the server.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h  UTL_TCP.CONNECTION;
  n  PLS_INTEGER;
  d  INTEGER;
BEGIN
  h := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  d := UTL_TCP.WRITE_TEXT(h, 'hi', NULL);
  n := UTL_TCP.AVAILABLE(h, 1);
  DBMS_OUTPUT.PUT_LINE('AVAILABLE=' || n);
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.WRITE_LINE

The procedure sends a text string through a TCP connection and adds end-of-line character(s) (newline) at the end.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.
- data (VARCHAR2) – text string to send (without end-of-line characters).

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Sending commands to interactive protocols where each command ends with a newline (TELNET, SMTP, FTP).
- Transmitting logs in a format where each entry is a separate line.
- Simplifying code when needing to send strings with terminating newline.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h    UTL_TCP.CONNECTION;
  sent PLS_INTEGER;
  d    INTEGER;
BEGIN
  h    := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  sent := UTL_TCP.WRITE_LINE(h, 'hello');
  DBMS_OUTPUT.PUT_LINE('WRITE_LINE: sent=' || sent);
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.GET_LINE

The function reads one text line from a TCP connection. Reading stops upon receiving end-of-line characters (newline) or reaching timeout.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.
- remove_crlf (BOOLEAN) – flag indicating whether to remove end-of-line characters from the result (default TRUE).

Output parameters:

- Returns the read line (without end-of-line characters if remove_crlf=TRUE).

Typical usage scenarios:

- Reading server responses line by line (e.g., reading HTTP headers or SMTP responses).
- Interactive interaction with a socket server that sends data line by line.
- Parsing text protocols where data is separated by newlines.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h    UTL_TCP.CONNECTION;
  line VARCHAR2(4000);
  s    INTEGER;
BEGIN
  h := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  s := UTL_TCP.WRITE_LINE(h, 'hello-line');
  line := UTL_TCP.GET_LINE(h, TRUE, FALSE);
  DBMS_OUTPUT.PUT_LINE('GET_LINE="' || line || '"');
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.WRITE_RAW

The procedure sends binary data (RAW) through an open TCP connection.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.
- data (RAW) – binary data to send.
- len (PLS_INTEGER) – number of bytes to send. If NULL, sends all data.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Sending files, images, or any binary data through a socket.
- Transmitting encrypted or compressed data.
- Implementing custom binary protocols over TCP.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h    UTL_TCP.CONNECTION;
  r    RAW(2000);
  sent PLS_INTEGER;
BEGIN
  h    := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  r    := HEXTORAW('48656C6C6F'); -- "Hello"
  sent := UTL_TCP.WRITE_RAW(h, r, NULL);
  DBMS_OUTPUT.PUT_LINE('WRITE_RAW: sent=' || sent);
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.GET_RAW

The function reads binary data (RAW) from a TCP connection.

Input parameters:

- c (PLS_INTEGER) – connection descriptor.
- len (PLS_INTEGER) – maximum number of bytes to read. If NULL, reads all available data until timeout.

Output parameters:

RAW – read binary data.

Typical usage scenarios:

- Receiving binary files downloaded via protocols (HTTP, FTP).
- Reading binary responses from services (e.g., receiving an image or PDF).
- Working with protocols that transmit data in mixed format (text + binary attachments).

Query examples:


```
SET SERVEROUTPUT ON
DECLARE
  h UTL_TCP.CONNECTION;
  r RAW(2000);
  s INTEGER;
BEGIN
  h := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  s := UTL_TCP.WRITE_RAW(h, HEXTORAW('DEADBEEF'), NULL);
  r := UTL_TCP.GET_RAW(h, 4);
  DBMS_OUTPUT.PUT_LINE('GET_RAW=' || RAWTOHEX(r));
  UTL_TCP.CLOSE_CONNECTION_P(h);
END;
/
```

UTL_TCP.CLOSE_ALL_CONNECTIONS

The procedure forcibly closes all open TCP connections created in the current session.

Input parameters:

No parameters.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Emergency cleanup of network resources upon critical error in the application.
- Guaranteed release of connections before ending a session or procedure.
- Use in exception blocks to prevent connection leaks.

Query examples:

```
SET SERVEROUTPUT ON
DECLARE
  h1 UTL_TCP.CONNECTION;
  h2 UTL_TCP.CONNECTION;
BEGIN
  h1 := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  h2 := UTL_TCP.OPEN_CONNECTION('0.0.0.0', 5678);
  UTL_TCP.CLOSE_ALL_CONNECTIONS_P();
  DBMS_OUTPUT.PUT_LINE('CLOSE_ALL_CONNECTIONS: ok');
END;
/
```

15.2.14. UTL_MAIL Package

UTL_MAIL.SEND

The procedure sends an email through the built-in Oracle mail server.

Input parameters:

- sender (VARCHAR2) – sender’s email address.
- recipients (VARCHAR2) – recipient address or list of addresses (comma-separated).
- cc (VARCHAR2) – copy address or list of addresses.
- bcc (VARCHAR2) – blind copy address or list of addresses.
- subject (VARCHAR2) – email subject.
- message (VARCHAR2) – message text.
- mime_type (VARCHAR2) – MIME type of content.
- priority (PLS_INTEGER) – email priority (1-high, 3-normal, 5-low).
- replyto (VARCHAR2) – reply-to address.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Automatic sending of notifications from triggers and stored procedures.
- Distribution of reports and processing results.
- Alerting administrators about critical events in the DB.

- Sending generated documents and forms.

Query examples:

```
SET SERVEROUTPUT ON
-- EXAMPLE: SENDING A REGULAR TEXT EMAIL
BEGIN
DBMS_OUTPUT.PUT_LINE('Sending regular text email...');
UTL_MAIL.SEND(
'sender@local', -- sender
'rcpt@local', -- recipients
NULL, NULL, -- cc, bcc
'No attachments', -- subject
'Hello! This is a test email.',-- email body
'text/plain; charset=UTF-8', -- MIME type
3, -- priority
NULL -- reply address
);
DBMS_OUTPUT.PUT_LINE('Email sent successfully');
END;
/
-- Result:
-- Sending regular text email...
-- Email sent successfully
```

UTL_MAIL.SEND_ATTACH_VARCHAR2

Procedure sends an email with a text attachment via the built-in Oracle mail server. The attachment is passed as a VARCHAR2 string.

Incoming parameters:

- sender (VARCHAR2) – sender’s email address.
- recipients (VARCHAR2) – recipient address or list of recipient addresses (comma-separated).
- cc (VARCHAR2) – address or list of addresses for CC (carbon copy).
- bcc (VARCHAR2) – address or list of addresses for BCC (blind carbon copy).
- subject (VARCHAR2) – email subject.
- message (VARCHAR2) – message text.
- mime_type (VARCHAR2) – MIME type of the email content.

- `priority (PLS_INTEGER)` – email priority (1-high, 3-normal, 5-low).
- `attachment (VARCHAR2)` – attachment content as a VARCHAR2 string.
- `inline (VARCHAR2)` – flag indicating whether the attachment is inline (TRUE) or a separate file (FALSE).
- `att_mime_type (VARCHAR2)` – attachment MIME type.
- `att_filename (VARCHAR2)` – attachment file name.
- `att_last_line (VARCHAR2)` – additional attachment parameters.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Sending generated reports as text files.
- Transferring logs and procedure execution results.
- Distributing configuration files or scripts.
- Sending data in structured text formats (CSV, XML, JSON).

Example queries:

```
SET SERVEROUTPUT ON
-- EXAMPLE: SENDING WITH A TEXT ATTACHMENT (INLINE)
BEGIN
DBMS_OUTPUT.PUT_LINE('Sending with text attachment (inline)...');
UTL_MAIL.SEND_ATTACH_VARCHAR2(
'sender@local',
'recpt@local',
NULL, NULL,
'Text Attachment',
'Email text',
'text/plain; charset=UTF-8',
3,
'Inline content', -- attachment content (VARCHAR2)
TRUE, -- inline attachment = TRUE
'text/plain; charset=UTF-8', -- attachment MIME type
'inline.txt', -- file name
NULL -- encoding / additional parameters
);
DBMS_OUTPUT.PUT_LINE('Email with inline attachment sent');
END;
/
-- Result:
-- Sending with text attachment (inline)...
-- Email with inline attachment sent
```

UTL_MAIL.SEND_ATTACH_RAW

Procedure sends an email with a binary attachment via the built-in Oracle mail server. The attachment is passed in RAW format.

Incoming parameters:

- sender (VARCHAR2) – sender’s email address.
- recipients (VARCHAR2) – recipient address or list of recipient addresses (comma-separated).
- cc (VARCHAR2) – address or list of addresses for CC (carbon copy).
- bcc (VARCHAR2) – address or list of addresses for BCC (blind carbon copy).
- subject (VARCHAR2) – email subject.
- message (VARCHAR2) – message text.
- mime_type (VARCHAR2) – MIME type of the email content.
- priority (PLS_INTEGER) – email priority (1-high, 3-normal, 5-low).

- attachment (RAW) – binary attachment content in RAW format.
- inline (BOOLEAN) – flag indicating whether the attachment is inline (TRUE) or a separate file (FALSE).
- att_mime_type (VARCHAR2) – attachment MIME type.
- att_filename (VARCHAR2) – attachment file name.
- att_last_line (VARCHAR2) – additional attachment parameters.

Output parameters:

- The procedure does not return a value.

Typical usage scenarios:

- Sending binary files: images, PDF documents, archives.
- Transferring generated reports in binary format.
- Distributing files with encrypted data.
- Sending attachments containing arbitrary binary data.

Example queries:

```
SET SERVEROUTPUT ON
-- EXAMPLE: SENDING WITH A BINARY ATTACHMENT (RAW)
BEGIN
DBMS_OUTPUT.PUT_LINE('Sending with binary attachment...');
UTL_MAIL.SEND_ATTACH_RAW(
'sender@local',
'rcpt@local',
NULL, NULL,
'Binary Attachment',
'Email text',
'text/plain; charset=UTF-8',
3,
'DEADBEEF', -- binary content
FALSE, -- inline attachment = FALSE
'application/octet', -- MIME type
'deadbeef.bin', -- file name
NULL -- additional parameters/encoding
);
DBMS_OUTPUT.PUT_LINE('Email with binary attachment sent');
END;
/
-- Result:
-- Sending with binary attachment...
-- Email with binary attachment sent
```

15.2.15. UTL_URL Package

UTL_URL.ESCAPE

Function performs escaping of special characters in a URL according to the RFC 3986 standard. Replaces unsafe characters with their percent-encoded representation.

Incoming parameters:

- url (VARCHAR2) – source URL string to escape.
- escape_reserved_chars (BOOLEAN) – flag indicating whether to escape reserved characters (TRUE - escape, FALSE - do not escape).

Output parameters:

- Returns a string with the escaped URL.

Typical usage scenarios:

- Preparing URLs for HTTP requests.
- Escaping parameters in web links.
- Processing user input before using it in a URL.
- Creating correct web addresses with spaces and special characters.

Example queries:

```
SET SERVEROUTPUT ON
-- Example 1: Escaping spaces in a full URL
-- Spaces in the URL are replaced with %20. The FALSE parameter means escaping ALL
special characters
DECLARE
test_var VARCHAR2(10) := 'A';
input_var VARCHAR2(100) := 'utl_url.escape('http://www.acme.com/a url with
space.html', FALSE)';
descr_var VARCHAR2(20) := 'Entire URL';
result_var VARCHAR2(400);
BEGIN
result_var := UTL_URL.ESCAPE('http://www.acme.com/a url with space.html', FALSE);
DBMS_OUTPUT.PUT_LINE(test_var || ' | ' || input_var || ' | ' || descr_var || ' | ' ||
result_var);
END;
/
-- Result:
--A | utl_url.escape('http://www.acme.com/a url with space.html', FALSE) | Entire URL
| http://www.acme.com/a%20url%20with%20space.html
```

UTL_URL.UNESCAPE

Function performs decoding of escaped characters in a URL, converting the percent-encoded representation back to original characters according to the RFC 3986 standard.

Incoming parameters:

- url (VARCHAR2) – string with escaped URL to decode.
- charset (VARCHAR2) – character encoding for decoding (default 'UTF-8').

Output parameters:

- Returns a string with the decoded URL.

Typical usage scenarios:

- Processing received URL parameters in web applications.
- Decoding data from HTTP requests.
- Reading and processing escaped paths and links.
- Restoring original text from percent-encoded representation.

Example queries:

```
SET SERVEROUTPUT ON
-- Example 1: Decoding simple escaped text
-- Replacing %20 with space (basic decoding)
DECLARE
test_var VARCHAR2(10) := 'E';
input_var VARCHAR2(100) := 'utl_url.unescape(''abc%20def'', 'UTF-8')';
descr_var VARCHAR2(20) := 'UNESCAPE %20';
result_var VARCHAR2(400);
BEGIN
result_var := UTL_URL.UNESCAPE('abc%20def', 'UTF-8');
DBMS_OUTPUT.PUT_LINE(test_var || ' | ' || input_var || ' | ' || descr_var || ' | ' ||
result_var);
END;
/
-- Result:
--E | utl_url.unescape('abc%20def', 'UTF-8') | UNESCAPE %20 | abc def
```

15.2.16. PLVDATE Package

PLVDATE.ADD_BIZDAYS

Function adds a specified number of business days to a given date, excluding weekends (Saturday and Sunday).

Key capabilities:

- adds business days to the specified date
- excludes weekends (Saturday and Sunday) from the calculation
- supports both positive and negative day values
- does not consider holidays (only standard weekends)

Input parameters:

- `base_date` - base date (DATE type)
- `days_to_add` - number of business days to add (INTEGER type)

Output parameters:

DATE - resulting date after adding business days:

- Future date, if `days_to_add` > 0
- Past date, if `days_to_add` < 0
- Original date, if `days_to_add` = 0

Typical usage scenarios:

- calculating task completion dates considering only business days
- determining payment dates under contracts
- planning goods delivery deadlines
- computing dates for business processes

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := '2025-10-08';
    v_count INTEGER := 1;
    v_result DATE;
BEGIN
    v_result := PLVDATE.ADD_BIZDAYS(v_date, v_count);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

PLVDATE.BIZDAYS_BETWEEN

Function calculates the number of business days between two dates, excluding weekends (Saturday and Sunday).

Key capabilities:

- Counts the number of business days between two dates
- Excludes weekends (Saturday and Sunday) from the calculation

- Returns a positive or negative value depending on the order of dates
- Does not consider holidays (only standard weekends)

Input parameters:

- start_date - start date (DATE type)
- end_date - end date (DATE type)

Output parameters:

INTEGER - number of business days between dates: * Positive number, if end_date > start_date * Negative number, if end_date < start_date * 0, if dates are equal

Typical usage scenarios:

- Calculating duration of work processes
- Determining number of business days for payment
- Analyzing task completion deadlines
- Planning resources based on business days

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date_1 DATE := DATE '2025-10-08';
    v_date_2 DATE := DATE '2025-10-10';
    v_result INTEGER;
BEGIN
    v_result := PLVDATE.BIZDAYS_BETWEEN(v_date_1, v_date_2);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

PLVDATE.INCLUDE_START

Function determines whether to include the start date when calculating time intervals.

Key capabilities:

- Controls inclusion of the start date in interval calculations
- Used in conjunction with other date functions

- Supports various modes for calculating time periods

Input parameters:

include_flag (BOOLEAN) - flag for including the start date: * TRUE - start date is included in calculation * FALSE - start date is excluded from calculation

Output parameters:

- VOID - function sets a parameter for the current session

Typical usage scenarios:

- Precise calculation of time intervals
- Financial calculations with/without start date inclusion
- Determining task completion deadlines
- Logic for working with date periods

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date_1 DATE := DATE '2025-10-08';
    v_date_2 DATE := DATE '2025-10-10';
    v_result INTEGER;
    v_is_start_including BOOLEAN;
BEGIN
    -- It is also acceptable to use true in the include_start argument
    PLVDATE.INCLUDE_START;
    v_result := PLVDATE.BIZDAYS_BETWEEN(v_date_1, v_date_2);
    v_is_start_including := PLVDATE.INCLUDING_START;
    IF v_is_start_including THEN
        DBMS_OUTPUT.PUT_LINE('Including start day: ' || v_result);
    END IF;
    PLVDATE.NOINCLUDE_START;
    v_is_start_including := PLVDATE.INCLUDING_START;
    v_result := PLVDATE.BIZDAYS_BETWEEN(v_date_1, v_date_2);
    IF NOT v_is_start_including THEN
        DBMS_OUTPUT.PUT_LINE('Not including start day: ' || v_result);
    END IF;
END;
/
```

PLVDATE.INCLUDING_START

Function returns the current value of the flag that determines the inclusion of the start date in time interval calculations.

Key capabilities:

- Allows checking the current state of the start date inclusion flag
- Used for verifying time period calculation settings
- Works in conjunction with PLVDATE.INCLUDE_START

Input parameters:

- No input parameters

Output parameters:

BOOLEAN - current flag value: * TRUE - start date is included in calculation * FALSE - start date is excluded from calculation

Typical usage scenarios:

- Checking current date calculation settings
- Debugging time interval calculations
- Coordinating behavior of different date functions
- Logging time calculation parameters

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date_1 DATE := DATE '2025-10-08';
    v_date_2 DATE := DATE '2025-10-10';
    v_result INTEGER;
    v_is_start_including BOOLEAN;
BEGIN
    -- It is also acceptable to use true in the include_start argument
    PLVDATE.INCLUDE_START;
    v_result := PLVDATE.BIZDAYS_BETWEEN(v_date_1, v_date_2);
    v_is_start_including := PLVDATE.INCLUDING_START;
    IF v_is_start_including THEN
        DBMS_OUTPUT.PUT_LINE('Including start day: ' || v_result);
    END IF;
    PLVDATE.NOINCLUDE_START;
    v_is_start_including := PLVDATE.INCLUDING_START;
    v_result := PLVDATE.BIZDAYS_BETWEEN(v_date_1, v_date_2);
    IF NOT v_is_start_including THEN
        DBMS_OUTPUT.PUT_LINE('Not including start day: ' || v_result);
    END IF;
END;
/
```

PLVDATE.ISBIZDAY Function checks whether the specified date is a business day (considering only standard weekends - Saturday and Sunday).

Key capabilities:

- Checks if a date is a business day
- Automatically identifies weekends (Saturday and Sunday)
- Does not consider holidays (only standard weekends)
- Returns a boolean value (TRUE/FALSE)

Input parameters:

- check_date - date to check (DATE type)

Output parameters:

BOOLEAN: * TRUE - if the date is a business day (Monday-Friday) * FALSE - if the date is a weekend (Saturday or Sunday)

Typical usage scenarios:

- Checking dates before planning tasks
- Validating business days for processes
- Filtering data by business days
- Integration with other date functions

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_country VARCHAR2(64) := 'RUSSIA';
    v_date DATE := DATE '2025-01-01';
    v_isbizday BOOLEAN;
BEGIN
    PLVDATE.DEFAULT_HOLIDAYS(v_country);
    v_isbizday := PLVDATE.ISBIZDAY(v_date);
    IF v_isbizday THEN
        DBMS_OUTPUT.PUT_LINE('The specified date is a business day');
    END IF;
    IF NOT v_isbizday THEN
        DBMS_OUTPUT.PUT_LINE('The specified date is NOT a business day');
    END IF;
END;
/
```

PLVDATE.ISLEAPYEAR

Function checks whether the specified year is a leap year.

Key capabilities:

- Determines leap years according to the Gregorian calendar
- Works with years across a wide range
- Returns a boolean value (TRUE/FALSE)
- Does not require additional parameters

Input parameters:

- year - year to check (INTEGER type)

Output parameters:

BOOLEAN: * TRUE - if the year is leap * FALSE - if the year is not leap

Typical usage scenarios:

- Checking leap years for calendar calculations
- Determining number of days in February
- Validating dates in information systems
- Financial calculations considering year length

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date_1 DATE := DATE '2024-01-01';
    v_date_2 DATE := DATE '2023-01-01';
    v_is_leap BOOLEAN;
BEGIN
    v_is_leap := PLVDATE.ISLEAPYEAR(v_date_1);
    IF v_is_leap THEN
        DBMS_OUTPUT.PUT_LINE('Year in the specified date: ' || v_date_1 || ' - is a
    leap year');
    END IF;
    v_is_leap := PLVDATE.ISLEAPYEAR(v_date_2);
    IF NOT v_is_leap THEN
        DBMS_OUTPUT.PUT_LINE('Year in the specified date: ' || v_date_2 || ' - is not
    a leap year');
    END IF;
END;
/
```

PLVDATE.NEXT_BIZDAY

Function returns the next business day after the specified date (excluding weekends).

Key capabilities:

- Finds the next business day (Monday-Friday)
- If the input date is already a business day, returns the next business day
- Does not consider holidays (only standard weekends)
- Always returns a date in the future relative to the input date

Input parameters:

- input_date - source date (DATE type)

Output parameters:

- DATE - next business day after the specified date

Typical usage scenarios:

- Calculating task completion deadlines
- Determining process start dates
- Planning subsequent business days
- Automating work calendars

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2024-11-05';
    v_result DATE;
BEGIN
    v_result := PLVDATE.NEXT_BIZDAY(v_date);
    DBMS_OUTPUT.PUT_LINE('Next business day from the specified date: ' || v_result);
END;
/
```

PLVDATE.NOINCLUDE_START

Function disables inclusion of the start date in time interval calculations for the current session.

Key capabilities:

- Sets the flag to exclude the start date from calculations
- Affects functions working with time intervals
- Operates within the current session
- Is the antagonist of PLVDATE.INCLUDE_START(FALSE)

Input parameters:

- No input parameters

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Configuring interval calculations without including the start date
- Financial period calculations
- Determining task completion deadlines
- Synchronizing with application business logic

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date_1 DATE := DATE '2025-10-08';
    v_date_2 DATE := DATE '2025-10-10';
    v_result INTEGER;
    v_is_start_including BOOLEAN;
BEGIN
    -- It is also acceptable to use true in the include_start argument
    PLVDATE.INCLUDE_START;
    v_result := PLVDATE.BIZDAYS_BETWEEN(v_date_1, v_date_2);
    v_is_start_including := PLVDATE.INCLUDING_START;
    IF v_is_start_including THEN
        DBMS_OUTPUT.PUT_LINE('Including start day: ' || v_result);
    END IF;
    PLVDATE.NOINCLUDE_START;
    v_is_start_including := PLVDATE.INCLUDING_START;
    v_result := PLVDATE.BIZDAYS_BETWEEN(v_date_1, v_date_2);
    IF NOT v_is_start_including THEN
        DBMS_OUTPUT.PUT_LINE('Not including start day: ' || v_result);
    END IF;
END;
/
```

PLVDATE.PREV_BIZDAY

Function returns the previous business day before the specified date (excluding weekends).

Key capabilities:

- Finds the previous business day (Monday-Friday)
- If the input date is already a business day, returns the previous business day
- Does not consider holidays (only standard weekends)
- Always returns a date in the past relative to the input date

Input parameters:

- input_date - source date (DATE type)

Output parameters:

- DATE - previous business day before the specified date

Typical usage scenarios:

- Determining the last business day before an event
- Calculating task deadlines in reverse order
- Analyzing work periods in the past
- Adjusting reporting dates to business days

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2024-11-06';
    v_result DATE;
BEGIN
    v_result := PLVDATE.PREV_BIZDAY(v_date);
    DBMS_OUTPUT.PUT_LINE('Previous business day from the specified date: ' ||
v_result);
END;
/
```

PLVDATE.SET_NONBIZDAY

Function allows setting a specified date as a non-business day (weekend or holiday) within the current session.

Key capabilities:

- Adds custom non-business days to the current session
- Supports various input date formats
- Allows setting weekdays as regular weekends
- Affects calculations of all business day functions

Input parameters:

- date_expr - date or day of the week to set as non-business (DATE/TEXT type)
- is_recurring - optional recurring flag (only for weekdays)

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Declaring one-time holiday days
- Setting regular weekend days
- Configuring calendar for testing
- Accounting for local non-business days

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_bizday BOOLEAN;
BEGIN
    v_is_bizday := PLVDATE.ISBIZDAY(v_date);
    IF v_is_bizday THEN
        DBMS_OUTPUT.PUT_LINE('Specified date: ' || v_date || ' - is a business day');
    END IF;
    DBMS_OUTPUT.PUT_LINE('Setting specified date: ' || v_date || ' - as a NON-business
day');
    PLVDATE.SET_NONBIZDAY(v_date);
    v_is_bizday := PLVDATE.ISBIZDAY(v_date);
    IF NOT v_is_bizday THEN
        DBMS_OUTPUT.PUT_LINE('Specified date: ' || v_date || ' - is now NOT a business
day');
    END IF;
    DBMS_OUTPUT.PUT_LINE('Setting date: ' || v_date || ' - back as a business day');
    PLVDATE.UNSET_NONBIZDAY(v_date);
    v_is_bizday := PLVDATE.ISBIZDAY(v_date);
    IF v_is_bizday THEN
        DBMS_OUTPUT.PUT_LINE('Specified date: ' || v_date || ' - is again a business
day');
    END IF;
END;
/
```

PLVDATE.UNSET_NONBIZDAY

Function cancels the setting of a specified date or weekday as a non-business day within the current session.

Key capabilities:

- Reverses the action of PLVDATE.SET_NONBIZDAY for specified dates
- Supports the same input formats as SET_NONBIZDAY
- Allows canceling both one-time and recurring non-business days
- Restores standard behavior of business day calculation functions

Input parameters:

- date_expr - date or day of the week to cancel non-business status (DATE/TEXT type)
- is_recurring - optional recurring flag (only for weekdays)

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Canceling temporary holiday days
- Returning standard weekend days to business days
- Adjusting work calendar
- Testing alternative scenarios

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_bizday BOOLEAN;
BEGIN
    v_is_bizday := PLVDATE.ISBIZDAY(v_date);
    IF v_is_bizday THEN
        DBMS_OUTPUT.PUT_LINE('Specified date: ' || v_date || ' - is a business day');
    END IF;
    DBMS_OUTPUT.PUT_LINE('Setting specified date: ' || v_date || ' - as a NON-business
day');
    PLVDATE.SET_NONBIZDAY(v_date);
    v_is_bizday := PLVDATE.ISBIZDAY(v_date);
    IF NOT v_is_bizday THEN
        DBMS_OUTPUT.PUT_LINE('Specified date: ' || v_date || ' - is now NOT a business
day');
    END IF;
    DBMS_OUTPUT.PUT_LINE('Setting date: ' || v_date || ' - back as a business day');
    PLVDATE.UNSET_NONBIZDAY(v_date);
    v_is_bizday := PLVDATE.ISBIZDAY(v_date);
    IF v_is_bizday THEN
        DBMS_OUTPUT.PUT_LINE('Specified date: ' || v_date || ' - is again a business
day');
    END IF;
END;
/
```

PLVDATE.UNUSE_EASTER

Function disables consideration of Easter and Easter holidays in business day

calculations for the current session.

Key capabilities:

- Disables automatic consideration of Easter and related holidays
- Affects all business day functions
- Operates only within the current session
- Allows temporarily excluding Easter holidays from calculations

Input parameters:

- No input parameters

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Temporarily disabling Easter holidays for testing
- Calculations without considering floating religious holidays
- Comparing results with different calendar settings
- Adapting calculations to specific business rules

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_using_easter BOOLEAN;
    v_is_using_great_friday BOOLEAN;
BEGIN
    -- Easter and Good Friday are set as holidays by default.
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is a holiday');
    END IF;
    PLVDATE.UNUSE_EASTER;
    PLVDATE.UNUSE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF NOT v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is NOT a holiday');
    END IF;
    IF NOT v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is NOT a holiday');
    END IF;
    PLVDATE.USE_EASTER;
    PLVDATE.USE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is again a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is again a holiday');
    END IF;
END;
/
```

PLVDATE.UNUSE_GREAT_FRIDAY

Function disables consideration of Good Friday in business day calculations for the current session.

Key capabilities:

- Disables automatic consideration of Good Friday as a holiday
- Affects all business day functions
- Operates only within the current session
- Allows temporarily excluding this religious holiday from calculations

Input parameters:

- No input parameters

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Temporarily disabling religious holidays for testing
- Calculations without considering Good Friday
- Comparing results with different calendar settings
- Adapting calculations to non-religious business rules

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_using_easter BOOLEAN;
    v_is_using_great_friday BOOLEAN;
BEGIN
    -- Easter and Good Friday are set as holidays by default.
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is a holiday');
    END IF;
    PLVDATE.UNUSE_EASTER;
    PLVDATE.UNUSE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF NOT v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is NOT a holiday');
    END IF;
    IF NOT v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is NOT a holiday');
    END IF;
    PLVDATE.USE_EASTER;
    PLVDATE.USE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is again a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is again a holiday');
    END IF;
END;
/
```

PLVDATE.USE_EASTER

Function enables consideration of Easter and related Easter holidays in business day calculations for the current session.

Key capabilities:

- Enables automatic consideration of Easter and Easter holidays
- Affects all business day functions
- Operates only within the current session
- Allows temporarily considering floating religious holidays

Input parameters:

- No input parameters

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Including Easter holidays for calculations
- Testing functions considering religious holidays
- Comparing results with different calendar settings
- Adapting calculations to regional holidays

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_using_easter BOOLEAN;
    v_is_using_great_friday BOOLEAN;
BEGIN
    -- Easter and Good Friday are set as holidays by default.
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is a holiday');
    END IF;
    PLVDATE.UNUSE_EASTER;
    PLVDATE.UNUSE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF NOT v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is NOT a holiday');
    END IF;
    IF NOT v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is NOT a holiday');
    END IF;
    PLVDATE.USE_EASTER;
    PLVDATE.USE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is again a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is again a holiday');
    END IF;
END;
/
```

PLVDATE.USE_GREAT_FRIDAY

Function enables consideration of Good Friday in business day calculations for the current session.

Key capabilities:

- Enables automatic consideration of Good Friday as a holiday
- Affects all business day functions
- Operates only within the current session
- Allows temporarily considering this religious holiday

Input parameters:

- No input parameters

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Including Good Friday for calculations
- Testing functions considering religious holidays
- Comparing results with different calendar settings
- Adapting calculations to regional holidays

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_using_easter BOOLEAN;
    v_is_using_great_friday BOOLEAN;
BEGIN
    -- Easter and Good Friday are set as holidays by default.
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is a holiday');
    END IF;
    PLVDATE.UNUSE_EASTER;
    PLVDATE.UNUSE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF NOT v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is NOT a holiday');
    END IF;
    IF NOT v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is NOT a holiday');
    END IF;
    PLVDATE.USE_EASTER;
    PLVDATE.USE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is again a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is again a holiday');
    END IF;
END;
/
```

PLVDATE.USING_EASTER

Function checks whether Easter and Easter holidays are being considered in the current session.

Key capabilities:

- Allows checking the current state of Easter consideration
- Returns a boolean value (TRUE/FALSE)
- Used for debugging and checking settings

Input parameters:

- No input parameters

Output parameters:

BOOLEAN: * TRUE - if Easter consideration is enabled * FALSE - if Easter consideration is disabled

Typical usage scenarios:

- Checking current business day calculation settings
- Debugging date functions
- Conditional operations depending on settings
- Logging system parameters

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_using_easter BOOLEAN;
    v_is_using_great_friday BOOLEAN;
BEGIN
    -- Easter and Good Friday are set as holidays by default.
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is a holiday');
    END IF;
    PLVDATE.UNUSE_EASTER;
    PLVDATE.UNUSE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF NOT v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is NOT a holiday');
    END IF;
    IF NOT v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is NOT a holiday');
    END IF;
    PLVDATE.USE_EASTER;
    PLVDATE.USE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is again a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is again a holiday');
    END IF;
END;
/
```

PLVDATE.USING_GREAT_FRIDAY

Function checks whether Good Friday is being considered in business day calculations for the current session.

Key capabilities:

- Allows checking the current state of Good Friday consideration
- Returns a boolean value (TRUE/FALSE)
- Used for debugging and checking settings

Input parameters:

- No input parameters

Output parameters:

BOOLEAN: * TRUE - if Good Friday consideration is enabled * FALSE - if Good Friday consideration is disabled

Typical usage scenarios:

- Checking current holiday settings
- Debugging business day calculations
- Conditional logic depending on settings
- Validating system configuration

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_date DATE := DATE '2025-10-09';
    v_is_using_easter BOOLEAN;
    v_is_using_great_friday BOOLEAN;
BEGIN
    -- Easter and Good Friday are set as holidays by default.
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is a holiday');
    END IF;
    PLVDATE.UNUSE_EASTER;
    PLVDATE.UNUSE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF NOT v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is NOT a holiday');
    END IF;
    IF NOT v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is NOT a holiday');
    END IF;
    PLVDATE.USE_EASTER;
    PLVDATE.USE_GREAT_FRIDAY;
    v_is_using_easter := PLVDATE.USING_EASTER;
    v_is_using_great_friday := PLVDATE.USING_GREAT_FRIDAY;
    IF v_is_using_easter THEN
        DBMS_OUTPUT.PUT_LINE('Easter is again a holiday');
    END IF;
    IF v_is_using_great_friday THEN
        DBMS_OUTPUT.PUT_LINE('Good Friday is again a holiday');
    END IF;
END;
/
```

15.2.17. PLVSTR Package

PLVSTR.BETWN

Function extracts a substring between specified delimiters from the source string.

Key capabilities:

- Extracts part of a string between two specified delimiters
- Supports multiple occurrences of delimiters
- Allows specifying starting search position
- Returns NULL if delimiters are not found

Input parameters:

- string_in - source string (TEXT)
- start_delim - start delimiter (TEXT)
- end_delim - end delimiter (TEXT)
- start_pos - optional starting search position (INTEGER, default 1)
- nth_appearance - optional occurrence number (INTEGER, default 1)

Output parameters:

- TEXT - found substring or NULL

Typical usage scenarios:

- Parsing structured text data
- Extracting values from delimited strings
- Processing logs and system messages
- Analyzing text templates

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var    VARCHAR2(100) := 'Harry and Sally are very happy';
    p_start  VARCHAR2(1)    := 'a';
    p_end    VARCHAR2(1)    := 'y';
    p_nth    INTEGER        := 2;
    p_skip   INTEGER        := 1;
BEGIN
    v_var := PLVSTR.BETWN(v_var, p_start, p_end, p_nth, p_skip);
    DBMS_OUTPUT.PUT_LINE(v_var);
END;
/
```

```
-- Extract domain from email between 1st '@' and 1st '.' skipping 0 occurrences
DECLARE
    v_email VARCHAR2(100) := 'user@sub.domain.example.com';
    v_result VARCHAR2(100);
BEGIN
    v_result := PLVSTR.BETWN(v_email, '@', '.', 1, 1);
    DBMS_OUTPUT.PUT_LINE('Domain: ' || v_result);
END;
/
```

PLVSTR.SWAP

Function returns the result of replacing a substring in the source string.

_Incoming parameters:

- str – source string
- substr – characters to substitute
- start – starting index of substring in the source string
- len – number of characters to replace

Output parameters:

- Function returns a value of type text.

Typical usage scenarios:

- Quick editing of strings with character replacement.
- Correcting typos in large texts or documents.
- Transforming data for easier further processing.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var_text VARCHAR2(10) := 'cafe';
    v_var_replace VARCHAR2(2) := 't';
    v_var_start INTEGER := 3;
    v_var_len INTEGER := 2;
    v_result VARCHAR2(10);
BEGIN
    v_result := PLVSTR.SWAP(v_var_text, v_var_replace, v_var_start, v_var_len);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_old_date VARCHAR2(20) := '2023-12-31';
    v_new_date VARCHAR2(20);
BEGIN
    v_new_date := PLVSTR.SWAP(v_old_date, '2024', 1, 4);
    DBMS_OUTPUT.PUT_LINE('New date: ' || v_new_date);
END;
/
```

PLVSTR.RIGHT

Function returns a specified number of characters from the end of the string.

Incoming parameters:

- str – source string
- num – number of characters to be included from the right side, or excluded from the source string from the left side (with negative value)

Output parameters:

- Function returns a value of type text.

Typical usage scenarios:

- Getting file extension from path.
- Extracting last characters from email addresses or identifiers.
- Normalizing strings, removing unwanted leading characters.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_str VARCHAR2(30) := 'abcde';
    v_pos INTEGER := 2;
    v_result VARCHAR2(30);
BEGIN
    v_result := PLVSTR.RIGHT(v_str, v_pos);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
-- Extracting last 4 digits of a card number
DECLARE
    v_card VARCHAR2(20) := '1234567812345678';
    v_last_digits VARCHAR2(10);
BEGIN
    v_last_digits := PLVSTR.RIGHT(v_card, 4);
    DBMS_OUTPUT.PUT_LINE('Last digits: ' || v_last_digits);
END;
/
```

PLVSTR.LEFT

Function returns a specified number of characters from the beginning of the string.

_Incoming parameters::

- str - source string
- num – number of characters to be returned from the left side, or excluded from the source string from the right side (with negative value)

Output parameters:

- Function returns a value of type text.

Typical usage scenarios:

- Extracting first characters of an email address or phone number.
- Normalizing strings, removing extra characters at the end.

- Getting file prefix from path.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var VARCHAR2(5) := 'abcde';
    v_pos INTEGER := 2;
    v_result VARCHAR2(10);
BEGIN
    v_result := PLVSTR.LEFT(v_var, v_pos);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_company VARCHAR2(50) := 'HollyWood&Sons';
    v_abbr VARCHAR2(10);
BEGIN
    v_abbr := PLVSTR.LEFT(v_company, 4);
    DBMS_OUTPUT.PUT_LINE('Abbreviation: ' || v_abbr);
END;
/
```

PLVSTR.RVRS

Function returns the reverse representation of a string with the ability to specify a range of indices.

Input parameters:

- str - source string;
- start - starting index of the source string;
- end - ending index of the source string.

Output parameters:

- Function returns a value of type text.

Typical usage scenarios:

- Checking palindromes (words that read the same forward and backward).
- Reverse sorting text for special requirements (e.g., cryptography).
- Testing algorithms for processing reversed strings.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var_text VARCHAR2(64) := 'Was it a car or a cat I saw';
    v_var_start INTEGER := 1;
    v_result VARCHAR2(64);
BEGIN
    v_result := PLVSTR.RVRS(v_var_text, v_var_start);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_word VARCHAR2(50) := 'A Santa at NASA';
    v_encrypted VARCHAR2(50);
BEGIN
    v_encrypted := PLVSTR.RVRS(v_word, 1);
    DBMS_OUTPUT.PUT_LINE('Encrypted: ' || v_encrypted);
END;
/
```

PLVSTR.RSTRIP

Function removes a specified number of substring occurrences from the end of the source string. If the number of occurrences is less than 1 or the substring is absent, the source string is returned.

Incoming parameters:

- str — source string; substr — substring to remove; num — number of substring occurrences to remove.

Output parameters:

- Function returns a value of type text.

Typical usage scenarios:

- Removing suffixes in file names.
- Processing paths and links to eliminate extra segments.
- Normalizing data with repeating parts at the end of a string.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var_first VARCHAR2(15) := 'prefix_suffix_';
    v_var_suf VARCHAR2(10) := 'suffix_';
    v_num INTEGER := 1;
    v_result VARCHAR2(10);
BEGIN
    v_result := PLVSTR.RSTRIP(v_var_first, v_var_suf, v_num);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_filename VARCHAR2(100) := 'database.backup.bak';
    v_clean_name VARCHAR2(100);
BEGIN
    v_clean_name := PLVSTR.RSTRIP(v_filename, '.bak', 1);
    DBMS_OUTPUT.PUT_LINE('Clean name: ' || v_clean_name);
END;
/
```

PLVSTR.LSTRIP

Function removes a specified number of substring occurrences from the beginning of the source string. If the number of occurrences is less than 1 or the substring is absent, the source string is returned.

Input parameters:

- str — source string;
- substr — substring to remove;
- num — number of substring occurrences to remove.

Output parameters:

- Function returns a value of type text.

Typical usage scenarios:

- Removing prefixes in file names or paths.
- Normalizing text data with repeating parts at the beginning of a string.
- Removing leading characters or service elements.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var_first VARCHAR2(15) := 'prefix_suffix_';
    v_var_pref VARCHAR2(10) := 'prefix_';
    v_num INTEGER := 1;
    v_result VARCHAR2(10);
BEGIN
    v_result := PLVSTR.LSTRIP(v_var_first, v_var_pref, v_num);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_url VARCHAR2(100) := 'https://www.example.com/page';
    v_clean_url VARCHAR2(100);
BEGIN
    v_clean_url := PLVSTR.LSTRIP(v_url, 'https://', 1);
    DBMS_OUTPUT.PUT_LINE('Without protocol: ' || v_clean_url);
END;
/
```

PLVSTR.RPART

Function returns a substring containing part of the string up to the specified occurrence of a delimiter character from the end of the string. Returns NULL or the source string if the substring is not found, depending on the `all_if_notfound` parameter.

Incoming parameters:

- `str` — source string;
- `div` — substring to search for;
- `start` — starting search index;
- `nth` — occurrence number of the substring to search for;
- `all_if_notfound` — logical value indicating whether to return the source string if `div` is not found.

Outgoing parameters:

- Function returns a value of type `text`.

Typical usage scenarios:

- Extracting domain from URL.
- Separating folder names from file paths.
- Splitting complex text into components by delimiters.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var_first VARCHAR2(15) := 'abracadabra';
    v_var_div VARCHAR2(5) := 'b';
    v_start INTEGER := 1;
    v_result VARCHAR2(10);
BEGIN
    v_result := PLVSTR.RPART(v_var_first, v_var_div, v_start);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_filename VARCHAR2(100) := 'document.backup.txt';
    v_name VARCHAR2(100);
BEGIN
    v_name := PLVSTR.RPART(v_filename, '.', 1);
    DBMS_OUTPUT.PUT_LINE('File name: ' || v_name);
END;
/
```

PLVSTR.LPART

Function returns a substring containing part of the string up to the specified occurrence of a delimiter character from the beginning of the string. Returns NULL or the source string if the substring is not found, depending on the `all_if_notfound` parameter.

Incoming parameters:

- `str` — source string;
- `div` — substring to search for;
- `start` — starting search index;
- `nth` — occurrence number of the substring to search for;
- `all_if_notfound` — logical value indicating whether to return the source string if `div` is not found.

Output parameters:

- Function returns a value of type `text`.

Typical usage scenarios:

- Extracting username from email address.
- Processing hierarchical paths and splitting them into components.
- Parsing and normalizing text data by known delimiters.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var_first VARCHAR2(5) := 'abba';
    v_var_second VARCHAR2(5) := 'b';
    v_pos INTEGER := 1;
    v_nth INTEGER := 2;
    v_result VARCHAR2(10);
BEGIN
    v_result := PLVSTR.LPART(v_var_first, v_var_second, v_pos, v_nth);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_full_path VARCHAR2(100) := 'C:\Users\Ivan\Documents\file.txt';
    v_path VARCHAR2(100);
BEGIN
    v_path := PLVSTR.LPART(v_full_path, '\', 1, 4);
    DBMS_OUTPUT.PUT_LINE('Path: ' || v_path);
END;
/
```

PLVSTR.INSTR

Function returns the index of the start of a substring occurrence in a string. Returns 0 if the substring is absent.

Input parameters:

- str — source string;
- substr — substring to search for;
- start — starting search index;
- nth — occurrence number to search for.

Output parameters:

- Function returns a value of type int.

Typical usage scenarios:

- Extracting username from email address.
- Processing hierarchical paths and splitting them into components.
- Parsing and normalizing text data by known delimiters.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var    VARCHAR2(100) := 'abcd';
    v_substr VARCHAR2(2)    := 'ab';
    v_pos    INTEGER;
BEGIN
    v_pos := PLVSTR.INSTR(v_var, v_substr);
    DBMS_OUTPUT.PUT_LINE(v_pos);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_var    VARCHAR2(100) := 'abcd';
    v_substr VARCHAR2(2)    := 'ab';
    v_pos    VARCHAR2(30);
BEGIN
    v_pos := PLVSTR.INSTR(v_var, v_substr);
    DBMS_OUTPUT.PUT_LINE(v_pos);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_text VARCHAR2(100) := 'My friend is red';
    v_pos  VARCHAR2(30);
BEGIN
    v_pos := PLVSTR.INSTR(v_text, 'e', 1, 2);
    DBMS_OUTPUT.PUT_LINE('Position of second "e": ' || v_pos);
END;
/
```

PLVSTR.SUBSTR

Function returns a substring of the text string starting from the specified start index. Additionally supports specifying substring length len to limit the number of characters. *Input parameters:*

- str — string from which to extract substring;
- start — starting index of substring. Supports negative values for counting from the end of the string;

- **len** — length of extracted substring. If value < 1, returns NULL.

Output parameters:

- Function returns a value of type text.

Typical usage scenarios:

- Extracting a short snippet from a long string.
- Trimming excess text to reduce data volume.
- Standardizing strings by removing extra information.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    v_var_text VARCHAR2(10) := 'abc';
    v_var_start INTEGER := 2;
    v_var_len INTEGER := 1;
    v_result VARCHAR2(64);
BEGIN
    v_result := PLVSTR.SUBSTR(v_var_text, v_var_start, v_var_len);
    DBMS_OUTPUT.PUT_LINE(v_result);
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    v_date VARCHAR2(20) := '2024-03-15';
    v_month VARCHAR2(10);
BEGIN
    v_month := PLVSTR.SUBSTR(v_date, 6, 2);
    DBMS_OUTPUT.PUT_LINE('Month: ' || v_month);
END;
/
```

PLVSTR.IS_PREFIX

Function checks whether the string prefix is a prefix (beginning) of the string str. Additionally supports specifying case sensitivity cs for controlling comparison precision.

Input parameters:

- str — source string whose beginning is checked;
- prefix — string whose presence is checked at the beginning of str;
- cs — case sensitivity flag (true — case sensitive, false — case insensitive). Optional parameter, defaults to case sensitive.

Output parameters:

- Function returns a value of type bool (true, if prefix is the beginning of str, and false — otherwise).

Typical usage scenarios:

- Checking file extension compliance (e.g., .txt, .pdf).
- Validating input data (email, phone number).
- Determining data types by prefix (e.g., identifiers, product codes).
- Routing or filtering data based on their beginning.

Example queries:

```
SET SERVEROUTPUT ON
DECLARE
    case_sensitivity BOOLEAN := TRUE;
    is_prefix BOOLEAN;
    v_var_first VARCHAR2(1) := 'A';
    v_var_second VARCHAR2(1) := 'a';
BEGIN
    is_prefix := PLVSTR.IS_PREFIX(v_var_first, v_var_second, case_sensitivity);
    IF is_prefix THEN
        DBMS_OUTPUT.PUT_LINE('Is a prefix');
    END IF;
    IF NOT is_prefix THEN
        DBMS_OUTPUT.PUT_LINE('Is not a prefix');
    END IF;
END;
/
```



```
SET SERVEROUTPUT ON
DECLARE
    is_prefix BOOLEAN;
    v_var_first VARCHAR2(1) := 'a';
    v_var_second VARCHAR2(1) := 'a';
BEGIN
    is_prefix := PLVSTR.IS_PREFIX(v_var_first, v_var_second);
    IF is_prefix THEN
        DBMS_OUTPUT.PUT_LINE('Is a prefix');
    END IF;
    IF NOT is_prefix THEN
        DBMS_OUTPUT.PUT_LINE('Is not a prefix');
    END IF;
END;
/
```

```
SET SERVEROUTPUT ON
DECLARE
    is_prefix BOOLEAN;
    v_phone VARCHAR2(20) := '+12125551234';
BEGIN
    is_prefix := PLVSTR.IS_PREFIX(v_phone, '+7');
    IF is_prefix THEN
        DBMS_OUTPUT.PUT_LINE('US/Canada number');
    ELSE
        DBMS_OUTPUT.PUT_LINE('International number');
    END IF;
END;
/
```

Note: The function also has overloads for working with numeric data types (int4, int8), which convert numbers to strings and then perform prefix checking.

PLVSTR.NORMALIZE

Function removes all invisible characters before and after the text.

Incoming parameters:

- str – input string;

Outgoing parameters:

- Function returns a value of type text. *Typical usage scenarios:*

- Normalizing data during import: cleaning data of extra spaces and non-printable characters.
- Preparing data for search: eliminating extra spaces before searching.
- Processing user input: cleaning text of extra characters before saving to database.

Example queries:

```
SET SERVEROUTPUT ON
SELECT '    text    ';
```

```
DECLARE v_str VARCHAR2(30) := ' text '; v_result VARCHAR2(30); BEGIN v_result :=
PLVSTR.NORMALIZE(v_str); DBMS_OUTPUT.PUT_LINE(v_result); END; /
```

```
SET SERVEROUTPUT ON
-- Normalizing address with extra spaces
DECLARE
    v_address VARCHAR2(200) := '    NEW    FETTER    LANE    EC4    ';
    v_normalized VARCHAR2(200);
BEGIN
    v_normalized := PLVSTR.NORMALIZE(v_address);
    DBMS_OUTPUT.PUT_LINE('Address: ' || v_normalized);
END;
/
```

NLS_CHARSET_NAME

Function **NLS_CHARSET_NAME** returns the name of the character set corresponding to the given numeric character set identifier.

Input Parameters:

- **number** (NUMBER): Numeric character set identifier.

Output Parameters:

- **Result:** A string of type **VARCHAR2** containing the character set name. If the passed number is not recognized as a valid character set identifier, the function returns **NULL**.

Typical Usage Scenarios:

- Determining the name of a character set from a known numeric identifier.
- Obtaining a list of available character sets in the database.

DBMS_SQL.CLOSE_CURSOR

Procedure **DBMS_SQL.CLOSE_CURSOR** closes the specified cursor and releases associated resources.

Input Parameters:

- **c** (NUMBER): The identification number of the cursor to be closed.

Output Parameters:

- The procedure does not return a value.

Typical Usage Scenarios:

- Completing work with a cursor after performing all necessary operations.
- Releasing resources occupied by a cursor to improve overall system performance.

DBMS_SQL.OPEN_CURSOR

Function **DBMS_SQL.OPEN_CURSOR** opens a new cursor intended for executing dynamic SQL queries.

Input Parameters:

- **treat_as_client_for_results** (BOOLEAN): Boolean value allowing self to be set as client for receiving nested query results (default is **FALSE**).
- **security_level** (NUMBER): Cursor security level (can be **NULL**, **1** or **2**; affects permission checking when executing SQL operations).

Output Parameters:

- Returns the identification number of the newly opened cursor.

Typical Usage Scenarios:

- Starting work with a new cursor for executing dynamic SQL queries.
- Possibility of reusing the same cursor for executing multiple queries while keeping it open.

DBMS_UTILITY.EXEC_DDL_STATEMENT

Procedure **DBMS_UTILITY.EXEC_DDL_STATEMENT** executes DDL commands (Data Definition Language commands such as **CREATE**, **ALTER**, **DROP**, **TRUNCATE**) directly from PL/SQL code.

Input Parameters:

- **ddl_string** (VARCHAR2): String containing the DDL command to be executed.

Output Parameters:

- The procedure does not return a value.

Typical Usage Scenarios:

- Executing dynamic DDL commands obtained during program execution.
- Generating and executing DDL queries that depend on program execution conditions (e.g., creating tables, indexes, or views at runtime).

DBMS_UTILITY.DB_VERSION

Function **DBMS_UTILITY.DB_VERSION** returns information about the version of the installed Oracle database.

Input Parameters:

- **ban** (PL/SQL TABLE): Boolean value indicating whether to return additional information about releases and patches.
- If **ban = TRUE**, release and patch numbers are returned.
- If **ban = FALSE**, only the main version number is returned.

Output Parameters:

- **release** (VARCHAR2): Main Oracle version (e.g., **19**).
- **ver_id** (VARCHAR2): Release number (e.g., **0**).
- **upd_id** (VARCHAR2): Update number (e.g., **0**).
- **patch_id** (VARCHAR2): Patch number (e.g., **0**).
- **port_upd_id** (VARCHAR2): Platform-specific update number (e.g., **0**).

Typical Usage Scenarios:

- Checking database version before performing migrations or installing updates.
- Using version information for decision making during PL/SQL code execution (e.g., supporting features of specific versions).

15.3. Description of Functions and Procedures

ADD_MONTHS

This function adds a specified number of months to a given date, returning a new date.

Main Capabilities:

- Adds or subtracts a specified number of months from a date
- Correctly handles year transitions
- Accounts for different numbers of days in months
- Supports negative values for subtracting months
- Special algorithm for handling last days of months

Input Parameters:

- date (DATE) - source date (DATE type)
- number_of_months (INTEGER) - integer number of months to add (positive or negative)

Output Parameters:

- Returns a new date (DATE type)
- For the last day of a month, returns the last day of the resulting month

Typical Usage Scenarios:

- Calculating contract and subscription expiration dates
- Determining periodic payment dates
- Generating reports for previous periods
- Financial calculations with monthly intervals
- Planning regular events

Query Examples:

```
-- Adding 3 months to a date  
SELECT ADD_MONTHS('2023-05-15', 3); -- Result: 2023-08-15
```

```
-- Subtracting 2 months from a date  
SELECT ADD_MONTHS('2023-05-31', -2); -- Result: 2023-03-31
```

```
-- Special case with last day of month  
SELECT ADD_MONTHS('2023-01-31', 1); -- Result: 2023-02-28
```

DECODE

This function implements conditional logic similar to the CASE operator using compact syntax.

Main Capabilities:

- Performs conditional value comparisons
- Supports multiple conditions
- Allows specifying a default value
- Works with various data types

Input Parameters:

- expression (ANY) - value to compare
- search1, result1 - first comparison pair and result
- searchN, resultN - additional comparison pairs
- default (ANY) - optional value when no matches found

Output Parameters:

- Returns resultN when expression matches searchN
- Returns default if specified and no matches found
- Returns NULL if no matches and default not specified

Typical Usage Scenarios:

- Converting codes to understandable values

- Simplifying complex conditions
- Processing various statuses
- Formatting data for reports

Query Examples:

```
-- Simple number comparison  
SELECT DECODE(1, 1, 'One', 'Other'); -- Result: 'One'
```

```
-- Multiple conditions  
SELECT DECODE(2, 1, 'January', 2, 'February', 3, 'March', 'Other'); -- Result:  
'February'
```

LAST_DAY

Function returns the last day of the month for the specified date.

Main Capabilities:

- Returns the last day of the month for the specified date
- Correctly handles leap years
- Works with various date types
- Supports boundary date values

Input Parameters:

- date (DATE) - source date (DATE or TIMESTAMP type)

Output Parameters:

- Returns a date (DATE type) - the last day of the month
- Preserves time part for TIMESTAMP

Typical Usage Scenarios:

- Calculating end dates of reporting periods
- Determining number of days in a month
- Generating payment calendars

- Processing time intervals

Query Examples:

```
-- Basic example  
SELECT LAST_DAY('2023-05-15'); -- Result: 2023-05-31
```

```
-- February in leap year  
SELECT LAST_DAY('2024-02-15'); -- Result: 2024-02-29
```

```
-- With TIMESTAMP  
SELECT LAST_DAY('2023-11-15 14:30:00'); -- Result: 2023-11-30
```

```
-- Boundary values  
SELECT LAST_DAY('2023-01-31'); -- Result: 2023-01-31
```

LNNVL

Function evaluates a condition and returns TRUE if the condition is false or NULL, and FALSE if the condition is true.

Main Capabilities:

- Handles conditions including NULL values
- Returns TRUE for false or NULL conditions
- Returns FALSE only for true conditions
- Useful for simplifying checks considering NULL

Input Parameters:

- condition (BOOLEAN) - logical expression to evaluate

Output Parameters:

BOOLEAN: * TRUE, if condition is false or NULL * FALSE, if condition is true

Typical Usage Scenarios:

- Filtering data considering NULL values

- Simplifying complex conditions with IS NULL
- Handling undefined values in conditions
- Creating inverted conditions with NULL safety

Query Examples:

```
-- Basic examples
SELECT LNNVL(1 = 2); -- TRUE (condition false)
SELECT LNNVL(1 = 1); -- FALSE (condition true)
SELECT LNNVL(NULL = 1); -- TRUE (condition NULL)
-- Range checking
SELECT LNNVL(10 BETWEEN 5 AND 15); -- Result: FALSE
SELECT LNNVL(20 BETWEEN 5 AND 15); -- Result: TRUE
```

MONTHS_BETWEEN

Function calculates the number of months between two dates including fractional part.

Main Capabilities:

- Calculates exact number of months between dates
- Returns fractional value for incomplete months
- Accounts for different numbers of days in months
- Supports negative values (when date1 < date2)

Input Parameters:

- date1 (DATE) - first date (DATE type)
- date2 (DATE) - second date (DATE type)

Output Parameters:

NUMBER - number of months between dates: * Positive, if date1 > date2 * Negative, if date1 < date2 * 0, if dates are equal

Typical Usage Scenarios:

- Calculating exact work experience
- Determining age in months

- Analyzing time intervals between events
- Financial calculations with monthly periodicity

Query Examples:

```
-- Complete months  
SELECT MONTHS_BETWEEN('2023-06-15', '2023-03-15'); -- Result: 3
```

```
-- Fractional months  
SELECT MONTHS_BETWEEN('2023-06-20', '2023-03-15'); -- Result: 3.16129 (03 months 5  
days)
```

```
-- Negative value  
SELECT MONTHS_BETWEEN('2023-03-15', '2023-06-20'); -- Result: -3.16129
```

```
-- Last days of months  
SELECT MONTHS_BETWEEN('2023-02-28', '2023-01-31'); -- Result: 1
```

NEXT_DAY

Function returns the date of the next specified day of the week after the given date.

Main Capabilities:

- Finds the next specified day of the week after the given date
- Supports various formats for specifying day of week
- Correctly handles transitions between months and years
- Preserves original time for TIMESTAMP

Input Parameters:

- date (DATE) - source date (DATE or TIMESTAMP type)
- day_of_week (VARCHAR2 | NUMBER) - day of week (string or number):
- String: 'MONDAY', 'TUESDAY', etc. (case insensitive)
- Number: 1=Sunday, 2=Monday,...,7=Saturday

Output Parameters:

- DATE - date of the next specified day of week
- For TIMESTAMP returns DATE (time discarded)

Typical Usage Scenarios:

- Planning regular events
- Calculating payment dates and deadlines
- Generating event calendars
- Automating reminders

Query Examples:

```
-- Next Monday  
SELECT NEXT_DAY('2023-05-15', 'MONDAY'); -- Result: 2023-05-22
```

```
-- Next Friday (numeric format)  
SELECT NEXT_DAY('2023-05-15', 6); -- Result: 2023-05-19
```

```
-- If day matches the sought day  
SELECT NEXT_DAY('2023-05-15', 'MONDAY'); -- Result: 2023-05-22 (15.05 - Monday)
```

```
-- Transition between months  
SELECT NEXT_DAY('2023-05-30', 'SATURDAY'); -- Result: 2023-06-03
```

NLSSORT

Function returns binary representation of a string for sorting according to specified locale rules.

Main Capabilities:

- Supports string sorting according to rules of various languages
- Ensures correct ordering of national alphabets
- Allows considering regional sorting features
- Supports various string comparison algorithms

Input Parameters:

- string (VARCHAR2 | CHAR) - string to sort (VARCHAR2 or CHAR type)
- nls_param (VARCHAR2) - optional locale parameter in format: 'sort_name' For example: 'BINARY_AI'

Output Parameters:

- RAW - binary representation of string for sorting
- NULL, if input string is NULL

Typical Usage Scenarios:

- Sorting data in national languages
- Creating indexes for language-dependent search
- Comparing strings considering regional features
- Optimizing queries with language-dependent conditions

Query Examples:

```
-- Sorting ENGLISH alphabet
SELECT * FROM (SELECT 'apple' as word UNION SELECT 'apricot' UNION SELECT 'banana')
ORDER BY NLSSORT(word, 'ENGLISH');
```

```
-- Comparison considering locale
SELECT CASE
    WHEN NLSSORT('hedgehog', 'ENGLISH') > NLSSORT('hedgehog', 'ENGLISH')
    THEN 'Yes' ELSE 'No'
END AS result;
```

NVL

Function replaces NULL values with a specified default value.

Main Capabilities:

- Replaces NULL with specified value
- Works with any data types
- Supports nested calls
- Preserves data type of original expression

Input Parameters:

- `expr (ANY)` - expression to check
- `default_value (ANY)` - default value

Output Parameters:

- Returns `expr`, if it is not NULL
- Returns `default_value`, if `expr` is NULL

Typical Usage Scenarios:

- Filling missing values
- Preventing errors when working with NULL
- Formatting data output
- Processing aggregate function results

Query Examples:

```
-- Replacing NULL with text  
SELECT NVL(NULL, 'No data'); -- Result: 'No data'
```

```
-- Replacing NULL with number  
SELECT NVL(NULL, 0) + 10; -- Result: 10
```

```
-- With dates  
SELECT NVL(NULL, SYSDATE) FROM dual;
```

ROUND

Function rounds numbers or dates to specified precision.

Main Capabilities:

- Rounds numeric values to specified number of decimal places
- Supports rounding dates to various time units
- Works with negative rounding numbers

- Preserves data type of original value

Input Parameters:

- number (NUMBER) - number to round
- precision (INTEGER) - optional precision parameter (default 0)

For dates: * date (DATE) - date to round * format (VARCHAR2) - rounding unit ('YEAR', 'MONTH', 'DAY', etc.)

Output Parameters:

- For numbers: rounded number of same type
- For dates: date rounded to beginning of specified period

Typical Usage Scenarios:

- Financial calculations with specified precision
- Aggregating data by time periods
- Formatting calculation results
- Preparing report data

Query Examples:

```
-- Rounding numbers
SELECT ROUND(123.4567, 2);    -- Result: 123.46
SELECT ROUND(123.4567, -1);   -- Result: 120
SELECT ROUND(125, -1);        -- Result: 130
```

```
-- Rounding dates
SELECT ROUND(SYSDATE, 'MONTH'); -- Result: 01-APR-2023 (if current date is 15-MAR-2023)
SELECT ROUND(TO_DATE('2023-06-15'), 'YEAR'); -- Result: 01-JAN-2023
```

SUBSTR

Function extracts a substring from a string.

Main Capabilities:

- Extracts part of a string from specified position
- Supports negative positions (counting from end of string)
- Allows limiting length of extracted substring
- Works with various string data types

Input Parameters:

- string (VARCHAR2) - source string
- start_position (INTEGER) - starting position for extraction (1-based)
- length (INTEGER) - optional substring length

Output Parameters:

- VARCHAR2 - extracted substring
- NULL, if input string is NULL

Typical Usage Scenarios:

- Extracting parts of strings (codes, identifiers)
- Analyzing structured textual data
- Processing fixed string formats
- Preparing data for display

Query Examples:

```
-- Basic extraction
SELECT SUBSTR('Hello World', 7) FROM dual;      -- Result: 'World'
SELECT SUBSTR('Hello World', 7, 3) FROM dual;   -- Result: 'Wor'
```

```
-- Fixed formats
SELECT SUBSTR('20230315', 1, 4) || '-' ||
       SUBSTR('20230315', 5, 2) || '-' ||
       SUBSTR('20230315', 7, 2) FROM dual;      -- Result: '2023-03-15'
```

SYS_EXTRACT_UTC

Function converts timestamp with time zone to UTC timestamp (Coordinated Universal Time).

Main Capabilities:

- Converts TIMESTAMP WITH TIME ZONE to UTC
- Preserves precision to fractions of seconds
- Automatically accounts for daylight saving time
- Supports all standard time zones

Input Parameters:

- timestamp_tz (TIMESTAMP WITH TIME ZONE) - timestamp with time zone (TIMESTAMP WITH TIME ZONE type)

Output Parameters:

- TIMESTAMP - timestamp in UTC

Typical Usage Scenarios:

- Normalizing timestamps for storage
- Comparing timestamps from different time zones
- Preparing data for international systems
- Analyzing time data without locale binding

Query Examples:

```
-- Converting current time  
SELECT SYS_EXTRACT_UTC(CURRENT_TIMESTAMP) FROM dual;
```

```
-- Converting specific timestamp  
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2023-05-15 14:30:00 Europe/Moscow') FROM dual;
```

SYSDATE

Function returns current date and time of the database server.

Main Capabilities:

- Returns current system date and time
- Includes time with precision to seconds

- Requires no parameters
- Uses database server time zone

Input Parameters:

- No input parameters

Output Parameters:

- DATE - current server date and time

Typical Usage Scenarios:

- Recording operation times
- Comparing with current date
- Calculating time intervals
- Setting timestamps

Query Examples:

```
-- Getting current date  
SELECT SYSDATE FROM dual;
```

TO_CHAR

Function converts numbers, dates, and timestamps to string representation.

Main Capabilities:

- Converts numbers to formatted strings
- Formats dates and time according to given pattern
- Supports various language and regional settings
- Allows specifying custom output formats

Input Parameters:

- value (NUMBER | DATE | TIMESTAMP | INTERVAL) - value to convert (NUMBER, DATE, TIMESTAMP, etc.)
- format (VARCHAR2) - optional output format parameter

- nls_params (VARCHAR2) - optional language parameters

Output Parameters:

- VARCHAR2 - formatted string

Typical Usage Scenarios:

- Formatting report data
- Preparing data for export
- Converting numbers and dates for display
- Creating string representations of complex data

Query Examples:

```
-- Formatting numbers
SELECT TO_CHAR(1234.56, '999G999D99') FROM dual; -- Result: '1 234,56'
SELECT TO_CHAR(0.25, '99.99%') FROM dual;      -- Result: '.25%'
```

```
-- Formatting dates
SELECT TO_CHAR(SYSDATE, 'DD-MON-YYYY HH24:MI:SS') FROM dual;
SELECT TO_CHAR(SYSDATE, 'Day, DD Month YYYY') FROM dual;
```

TO_MULTI_BYTE

Function converts single-byte characters in a string to their multi-byte equivalents.

Main Capabilities:

- Converts single-byte characters to multi-byte
- Supports various character encodings
- Preserves original multi-byte characters unchanged
- Useful for ensuring data uniformity

Input Parameters:

- string (VARCHAR2) - source string for conversion

Output Parameters:

- VARCHAR2 - string with converted characters

Typical Usage Scenarios:

- Normalizing text data before comparison
- Preparing data for systems requiring multi-byte characters
- Ensuring uniformity in mixed character sets
- Converting data before export

Query Examples:

```
-- Basic conversion
SELECT TO_MULTI_BYTE('ABC123') FROM dual; -- Converts characters if possible
```

```
-- Mixed characters
SELECT TO_MULTI_BYTE('A C') FROM dual; -- Converts only single-byte 'A' and 'C'
```

TO_SINGLE_BYTE

Function converts multi-byte characters in a string to their single-byte equivalents where possible.

Main Capabilities:

- Converts multi-byte characters to single-byte
- Preserves original single-byte characters unchanged
- Supports various character encodings
- Useful for data normalization

Input Parameters:

- string (VARCHAR2) - source string for conversion

Output Parameters:

- VARCHAR2 - string with converted characters

Typical Usage Scenarios:

- Normalizing text data before comparison

- Preparing data for systems requiring single-byte characters
- Ensuring uniformity in mixed character sets
- Converting data before export

Query Examples:

```
-- Basic conversion
SELECT TO_SINGLE_BYTE('ABC123') FROM dual; -- Result: 'ABC123' (if equivalents exist)
```

```
-- Mixed characters
SELECT TO_SINGLE_BYTE('ABC') FROM dual;      -- Result: 'ABC' (if equivalents exist)
```

TRUNC

Function truncates numbers or dates to specified precision.

Main Capabilities:

- Truncates numeric values to specified number of decimal places
- Supports truncating dates to various time units
- Works with negative numbers for integer part truncation
- Preserves data type of original value

Input Parameters:

For numbers: * number (NUMBER) - number to truncate * precision (INTEGER) - optional precision parameter (default 0)
For dates: * date (DATE) - date to truncate * format (VARCHAR2) - truncation unit ('YEAR', 'MONTH', 'DAY', etc.)

Output Parameters:

- For numbers: truncated number of same type
- For dates: date truncated to beginning of specified period

Typical Usage Scenarios:

- Financial calculations with controlled precision
- Aggregating data by time periods

- Preparing data for grouping
- Removing fractional parts when necessary

Query Examples:

```
-- Truncating numbers
SELECT TRUNC(123.4567, 2);    -- Result: 123.45
SELECT TRUNC(123.4567, -1);  -- Result: 120
SELECT TRUNC(125, -1);       -- Result: 120
```

```
-- Truncating dates
SELECT TRUNC(SYSDATE, 'MONTH'); -- Result: 01-MAR-2023 (if current date is 15-MAR-2023)
SELECT TRUNC(TO_DATE('2023-06-15'), 'YEAR'); -- Result: 01-JAN-2023
```

NVL2

Extended NULL handling function that returns different values depending on presence/absence of NULL.

Main Capabilities:

- Returns one value for non-NULL data and another for NULL
- Supports all data types
- Provides more flexible logic than NVL
- Preserves data type of return value

Input Parameters:

- `expr (ANY)` - expression to check
- `not_null_value (ANY)` - value returned if `expr` is not NULL
- `null_value (ANY)` - value returned if `expr` is NULL

Output Parameters:

- Returns `not_null_value`, if `expr` is not NULL
- Returns `null_value`, if `expr` is NULL

Typical Usage Scenarios:

- Conditional data formatting
- Different handling of NULL and non-NULL values
- Forming complex conditions with NULL
- Replacing nested NVL with more readable code

Query Examples:

```
-- Basic example  
SELECT NVL2(NULL, 'Data exists', 'No data'); -- Result: 'No data'
```

TO_DATE

Function converts a string to a DATE type value according to the specified format.

Main Capabilities:

- Converts string representations of dates to DATE type
- Supports various custom date formats
- Provides handling of dates in different languages
- Performs date validity checking

Input Parameters:

- string (VARCHAR2)- string with date to convert
- format (VARCHAR2)- optional date format parameter

Output Parameters:

- DATE - converted date value

Typical Usage Scenarios:

- Importing dates from text files
- Converting custom date formats
- Processing international date formats
- Validating string representations of dates

Query Examples:

```
-- Conversion with explicit format  
SELECT TO_DATE('15-05-2023', 'DD-MM-YYYY') FROM dual;  
-- Result: 15-MAY-2023
```

```
-- Using different formats  
SELECT TO_DATE('20230515', 'YYYYMMDD') FROM dual;  
-- Result: 15-MAY-2023
```

16. Built-in Procedures and Functions of MS SQL Server Dialect

16.1. General Information

Stored procedures are blocks of code that allow performing specific actions in the database, such as data selection, insertion, updating, or deletion. Unlike regular SQL, stored procedures provide the ability to use procedural constructs such as conditions, loops, and exception handling.

The Digital DBMS Digital Q.DataBase supports stored procedures in the **T-SQL** dialect, developed by Microsoft for SQL Server. These procedures execute inside the database and can be called from SQL queries or client applications.

Why are stored procedures needed?

- 1. Task Automation:** Stored procedures allow programming frequently repeated operations, eliminating the need to perform the same actions manually.
- 2. Performance Increase:** Executing stored procedures inside the database is significantly faster than multiple database calls from external applications.
- 3. Security Enhancement:** Restricting data access through procedures reduces the likelihood of abuse and improves database security.
- 4. Logic Centralization:** All data processing logic is concentrated in the database, simplifying maintenance and modernization.

How do stored procedures work?

- 1. Procedure Creation:** A procedure is created using the `CREATE PROCEDURE` command, which describes its header, parameters, and procedure body.
- 2. Variable Declaration:** Inside a procedure, you can declare variables and use them for intermediate calculations.
- 3. Execution Flow Control:** Procedures support control constructs such as conditions (`IF-THEN-ELSE`), loops (`WHILE`, `BEGIN...END`), and exception handling (`TRY...CATCH`).
- 4. Procedure Call:** Procedures are called using the `EXEC` command or through SQL queries.

Example of a simple T-SQL procedure in Digital Q.DataBase:

```
CREATE PROCEDURE update_employee_salary
    @employee_id INT,
    @new_salary DECIMAL(10, 2)
AS
BEGIN
    UPDATE employees
    SET salary = @new_salary
    WHERE employee_id = @employee_id;
END;
```

The DBMS supports the following MS SQL SERVER syntax procedures (the list will be expanded in subsequent product releases):

1. fn_helpcollations
2. fn_listextendedproperty
3. sp_addlinkedsrvlogin
4. sp_addrole
5. sp_addrolemember
6. sp_changedbowner
7. sp_column_privileges
8. sp_columns
9. sp_columns_100
10. sp_columns_managed
11. sp_cursor_list
12. sp_databases
13. sp_datatype_info
14. sp_datatype_info_100
15. sp_describe_cursor
16. sp_dropextendedproperty
17. sp_droplinkedsrvlogin
18. sp_droprole
19. sp_droprolemember

20. sp_dropserver
21. sp_enum_oledb_providers
22. sp_fkeys
23. sp_getapplock
24. sp_helpdb
25. sp_helpdbfixedrole
26. sp_helplinkedsrvlogin
27. sp_helprole
28. sp_helprolemember
29. sp_helpsrvrolemember
30. sp_helpuser
31. sp_linkedservers
32. sp_oledb_ro_username
33. sp_pkeys
34. sp_procedure_params_100_managed
35. sp_releaseapplock
36. sp_renamedb
37. sp_serveroption
38. sp_special_columns
39. sp_special_columns_100
40. sp_sproc_columns
41. sp_sproc_columns_100
42. sp_statistics
43. sp_statistics_100
44. sp_stored_procedures
45. sp_table_privileges
46. sp_tablecollations_100
47. sp_tables
48. sp_updateextendedproperty

49. sp_updatestats
50. sp_who
51. sp_help
52. sp_bindefault
53. sp_unbindefault
54. sp_defaultdb
55. sp_dropuser
56. sp_adduser
57. sp_droptype
58. sp_addtype
59. sp_password
60. sp_spaceused
61. sp_primary_keys_rowset
62. sp_indexes_rowset
63. sp_unprepare
64. sp_prepare

The DBMS also supports the following MS SQL SERVER syntax functions

Date and Time

1. DATEFIRST
2. CURRENT_TIMESTAMP
3. DATE_BUCKET
4. DATEADD
5. DATEDIFF
6. DATEDIFF_BIG
7. DATEFROMPARTS
8. DATETIME2FROMPARTS
9. DATETIMEFROMPARTS
10. DATETIMEOFFSETFROMPARTS

- 11. DATETRUNC
- 12. DAY
- 13. EOMONTH
- 14. GETDATE
- 15. GETUTCDATE
- 16. ISDATE
- 17. MONTH
- 18. SYSDATETIME
- 19. SYSDATETIMEOFFSET
- 20. SYSUTCDATETIME
- 21. TIMEFROMPARTS
- 22. TODATETIMEOFFSET
- 23. YEAR

Logical

- 24. CHOOSE
- 25. GREATEST
- 26. IIF
- 27. LEAST

Mathematical

- 28. ABS
- 29. ACOS
- 30. ASIN
- 31. ATAN
- 32. ATN2
- 33. CEILING
- 34. COS
- 35. COT
- 36. COUNT

37. COUNT_BIG

38. DEGREES

39. EXP

40. FLOOR

41. LOG

42. LOG10

43. MAX

44. MIN

45. PI

46. POWER

47. RAND

48. ROUND

49. SIGN

50. SIN

51. SQRT

52. SQUARE

53. STDEV

54. STDEVP

55. SUM

56. TAN

57. VAR

58. VARP

Metadata

59. APP_NAME

60. COL_LENGTH

61. COL_NAME

62. COLUMNPROPERTY

63. DATABASEPROPERTYEX

- 64. DB_ID
- 65. NEXT VALUE FOR
- 66. OBJECT_DEFINITION
- 67. OBJECT_ID
- 68. OBJECT_NAME
- 69. OBJECT_SCHEMA_NAME
- 70. PARSENAME
- 71. SCHEMA_ID
- 72. SCHEMA_NAME
- 73. SCOPE_IDENTITY
- 74. SERVERPROPERTY
- 75. TYPE_ID
- 76. TYPE_NAME

String

- 77. ASCII
- 78. CHAR
- 79. CHARINDEX
- 80. CONCAT
- 81. FORMAT
- 82. LEFT
- 83. LEN
- 84. LOWER
- 85. LTRIM
- 86. N
- 87. PATINDEX
- 88. QUOTENAME
- 89. REPLACE
- 90. RIGHT

- 91. RTRIM
- 92. SPACE
- 93. STR
- 94. STRING_AGG
- 95. STRING_ESCAPE
- 96. SUBSTRING
- 97. TRANSLATE
- 98. TRIM
- 99. UPPER

16.2. Procedure Descriptions

fn_helpcollations

The fn_helpcollations procedure is designed to output a complete list of all available character sets (Collations) supported by the SQL Server.

Main Capabilities:

- Returns a complete list of all possible character sets (collations).
- Used for selecting the required character set when configuring a new server, database, table, or index.
- Shows descriptions of character sets, helping to choose the optimal option for the required localization or application features.

Input Parameters:

- No input parameters. The procedure returns a result table immediately upon execution.

Output Parameters:

Table with two columns: * CollationName: Full name of the character set. * Description: Detailed description of the character set, including regional features, sorting and comparison rules.

Typical Usage Scenarios:

- Determining character set for a new database or object.
- Analyzing existing character sets before data migration.
- Optimizing query performance by selecting appropriate character set.

Query Examples:

```
-- View all available character sets  
SELECT * FROM fn_helpcollations();
```

fn_listextendedproperty

The `fn_listextendedproperty` procedure is used to extract extended properties of one or more objects in a SQL Server database. Extended properties allow storing additional metadata about database objects, such as comments, versions, authorship, and other arbitrary data.

Main Capabilities:

- Extracts one or more extended properties of an object.
- Supports retrieving information for different object types (databases, schemas, tables, columns, functions, stored procedures, etc.).
- Can be used for auditing changes in database structure or adding documentation directly within the database schema.

Input Parameters:

- [@name] = property_name: Optional parameter specifying the name of the requested extended property. If value is not specified, returns all properties of the specified level.
- [@level0type] = level_0_type: Object hierarchy level (usually one of three levels: SCHEMA, TABLE, VIEW, PROCEDURE, FUNCTION, etc.). Required when subsequent levels are present.
- [@level0name] = level_0_name: Name of the first level object. For example, schema name or table/view name.
- [@level1type] = level_1_type: Additional detail level specifying the second level object. For tables, this could be a column or index.
- [@level1name] = level_1_name: Name of the second level object.

- [@level2type] = level_2_type: Third detail level (if applicable). Rarely used.
- [@level2name] = level_2_name: Name of the third level (rarely used).

Output Parameters:

Result table containing the following fields: * class_desc: Type of object to which the property belongs. * major_id: Identifier of the top-level object. * minor_id: Identifier of the nested element (e.g., column or index). * name: Name of the extended property. * value: Value of the extended property.

Typical Usage Scenarios:

- Adding comments and documentation directly to the database structure.
- Auditing changes in database schema and tracking versions.
- Automatically generating documentation using extended properties.

Query Examples:

```
-- List all extended properties for the entire database
SELECT * FROM fn_listextendedproperty(NULL, NULL, NULL, NULL, NULL, NULL, NULL);
```

```
-- Display information about extended property for table publishers
SELECT * FROM fn_listextendedproperty('MS_Description', 'SCHEMA', 'dbo', 'TABLE',
'publishers');
```

sp_addlinkedsevrlogin

This procedure creates or updates a mapping between a login name on the local SQL Server instance and a security account on a remote server.

Main Capabilities:

- Creates a mapping between a local account and a remote account on a linked server.
- Allows configuring access rights for executing distributed queries and ensures security of communication between servers.
- Designed to simplify security management in systems with multiple SQL Server instances.

Input Parameters:

- [**@rmtsrvname** =] N'rmtsrvname': Name of the linked server to which the login mapping applies. Required parameter.
- [**@useself** =] 'useself': Flag determining connection method to remote server: using current user credentials or explicitly sending login and password. Possible values: true (use current credentials) or false (explicitly specify login and password). Default: true.
- [**@locallogin** =] N'locallogin': Name of local login used for mapping. Can be SQL Server or Windows login name. If set to NULL, mapping applies to all local logins. Default: NULL.
- [**@rmtuser** =] N'rmtuser': Name of remote login used to connect to linked server if @useself parameter equals false. Default: NULL.
- [**@rmtpassword** =] N'rmtpassword': Password for remote login specified in @rmtuser argument. Default: NULL.

Output Parameters:

- Return value: result code:
- 0: operation completed successfully.
- 1: error occurred.

Typical Usage Scenarios:

- Creating distributed queries accessing data on another server.
- Configuring trusted connections between servers for data migration.
- Managing user access rights when accessing objects on remote server.

Query Examples:

```
-- Connect all local login names to linked server Accounts  
EXEC sp_addlinkedsevrlogin 'QDATABASE', false;
```

```
-- Connect user Domain\Mary to linked server Accounts  
EXEC sp_addlinkedsevrlogin 'QDATABASE', 'false', NULL, 'MaryP', '<password>';
```

sp_addrole

This procedure creates a new database role in the current database.

Main Capabilities:

- Adds a new role to the current database.
- Assigns database role owner.
- Provides ability for further role membership assignment and permission granting.

Input Parameters:

- [**@rolename** =] N'rolename': Name of the database role being created. Must be a unique identifier 1-128 characters long. Backslash ("\") character not allowed.
- [**@ownername** =] N'ownername': Name of the owner of the new database role. Default is current user. Must be a valid database object (role, user, or group).

Output Parameters:

- Execution result:
- 0: operation completed successfully.
- 1: error occurred.

Typical Usage Scenarios:

- Creating a new role for subsequent user addition and privilege assignment.
- Simplified database security organization by grouping users into roles.

Query Examples:

```
-- Add new role Managers to current database  
EXEC sp_addrole 'Managers';
```

sp_addtype

Stored procedure sp_addtype allows creating data type aliases in the database.

Main Capabilities:

- Forms custom data types based on built-in SQL Server types.
- Simplifies work with data types, making them clearer and more convenient for development.

Input Parameters:

- [**@typename** =] N'typename': Name of the new data type.
- [**@phystype** =] N'phystype': Base data type on which the alias is created.
- [**@nulltype** =] 'nulltype': Method of handling NULL values (default or prohibition).
- [**@owner** =] N'owner': Owner is always considered to be schema dbo.

Output Parameters:

- Command execution result.
- Return code: 0 (success) or 1 (failure).

Features:

- Data types are stored in schema dbo.
- NULL value behavior can be set separately from base data type.

Query Examples:

```
-- Creating data type without NULL support  
EXECUTE sp_addtype ssn, 'varchar(11)', 'NOT NULL';
```

sp_adduser

Stored procedure sp_adduser creates a new user in the current database.

Main Capabilities:

- Registers a new user with ability to assign role and permissions.
- Used for expanding database participant circle.

Input Parameters:

- [**@loginame** =] N'loginame': User login name.
- [**@name_in_db** =] N'name_in_db': User name in the database (defaults to login name).
- [**@grpname** =] N'grpname': Database role in which the new user will be included.

Output Parameters:

- Informs about operation completion.
- Return code: 0 (success) or 1 (failure).

Features:

- New user automatically joins public group if role not specified.
- Requires database owner privileges.

Query Examples:

```
A. Adding user with role assignment  
EXECUTE sp_adduser 'Vidur', 'Vidur', 'Recruiting';
```

sp_bindefault

Stored procedure sp_bindefault is designed to bind a default value to a column or data type alias.

Main Capabilities:

- Sets default value for column or data type.
- Simplifies data initialization process when creating records.

Input Parameters:

- [@defname =] N'defname': Name of default value created by CREATE DEFAULT command.
- [@objname =] N'objname': Name of column or data type to which default value should be applied.
- [@futureonly =] 'futureonly': Optional parameter indicating that existing value is not changed, only future columns receive new default value.

Output Parameters:

- Informational messages about binding result.
- Return code: 0 (success) or 1 (failure).

Features:

- Does not work with data types such as timestamp, varchar(max) and other special

types.

- Requires permission to access objects.

Query Examples:

```
-- Binding default value to column  
EXECUTE sp_bindefault 'today', 'HumanResources.Employee.HireDate';
```

sp_changedbowner

Procedure sp_changedbowner changes the owner of the current database.

Main Capabilities:

- Changes database owner to specified login name.
- Transfers new owner's permissions to dbo role in current database.
- Convenient for quickly changing database owners when responsible persons change or duties are redistributed.

Input Parameters:

- [@loginame =] N'loginame': Login name of new database owner. Must correspond to valid SQL Server or Windows user. Required parameter.
- [@map =] 'map': This parameter is deprecated and kept only for supporting old scripts. No need to use it.

Output Parameters:

- Execution result:
- 0: successfully executed.
- 1: execution error.

Typical Usage Scenarios:

- Transferring database ownership to another employee or service.
- Quickly assigning new owner for administrative control.

Query Examples:

```
-- Change current database owner to Albert  
EXEC sp_changedbowner 'Albert';
```

sp_column_privileges

Procedure `sp_column_privileges` returns information about access rights to individual columns of the specified table in the current database.

Main Capabilities:

- Provides information about owners, privileges, and access features for each column of selected table.
- Helps determine user and role access rights to specific table columns.

Input Parameters:

- `[@table_name = ] N'table_name'`: Name of table for which column access rights information is needed. Required parameter.
- `[@table_owner = ] N'table_owner'`: Name of table owner. If not specified, returns first found table with that name visible to current user.
- `[@table_qualifier = ] N'table_qualifier'`: Name of table qualifier (e.g., database). Used for precise table location determination.
- `[@column_name = ] N'column_name'`: Specific column for which to return access rights. If not specified, returns access rights for all table columns.

Output Parameters:

- Returned result is a table with fields such as:
- Table name (TABLE_NAME)
- Column name (COLUMN_NAME)
- Access rights (PRIVILEGE)
- Who granted the right (GRANTOR)
- To whom rights are granted (GRANTEE)
- Ability to delegate rights (IS_GRANTABLE)

Typical Usage Scenarios:

- Checking user access level to individual table columns.

- Administering database security and auditing privileges.

Query Examples:

```
-- Get access rights to column Employee.SalariedFlag  
EXEC sp_column_privileges  
@table_name = 'test',  
@column_name = '%';
```

sp_columns

Procedure sp_columns returns information about columns of the specified table or view in the current database.

Main Capabilities:

- Provides detailed information about each column (data type, size, nullability, etc.) for selected objects.
- Suitable for analyzing database structure and developing design solutions.

Input Parameters:

- [**@table_name =]** N'table_name': Name of table or view for which column information is needed. Required parameter.
- [**@table_owner =]** N'table_owner': Name of table or view owner. If not specified, current user's schema is applied.
- [**@table_qualifier =]** N'table_qualifier': Name of qualifier (e.g., database name). Default is current database.
- [**@column_name =]** N'column_name': Limits selection to only specified column. If not set, returns all columns.
- [**@ODBCVer =]** ODBCVer: ODBC version used for result formation. Possible values: 2 (default) or 3.

Output Parameters:

- Table with results, including information about each column, such as:
- Column name (COLUMN_NAME)
- Data type (TYPE_NAME)
- Size (LENGTH)

- Nullability (NULLABLE)
- Column order (ORDINAL_POSITION)

Typical Usage Scenarios:

- Analyzing database structure when developing queries or optimized solutions.
- Assessing data structure compatibility when migrating data from other systems.

Query Examples:

```
-- Get information about columns of table Department  
EXEC sp_columns @table_name = 'Department';
```

sp_columns_managed

Procedure `sp_columns_managed` is similar to standard procedure `sp_columns`, but specifically designed for working with database objects managed by external systems (such as SQL Azure managed instances or autonomous services).

Main Capabilities:

- Provides information about columns of tables and views managed by external services.
- Supports work with cloud services and external data stores.

Input Parameters:

- Similar to `sp_columns` procedure parameters:
- `[@table_name = ] N'table_name':` Name of table or view.
- `[@table_owner = ] N'table_owner':` Name of object owner.
- `[@table_qualifier = ] N'table_qualifier':` Name of qualifier (e.g., database name).
- `[@column_name = ] N'column_name':` Limits selection to one column.
- `[@ODBCVer = ] ODBCVer:` ODBC version (optional parameter).

Output Parameters:

- Returns table with same characteristics as `sp_columns`, including:
- Column name (COLUMN_NAME)

- Data type (TYPE_NAME)
- Length (LENGTH)
- Nullability (NULLABLE)
- Ordinal position (ORDINAL_POSITION)

Typical Usage Scenarios:

- Working with cloud databases and virtualized environments.
- Conducting analysis of data structure located on managed resources.

Query Examples:

```
-- Get information about columns of table Employee  
EXEC sp_columns_managed 'Employee ';
```

sp_cursor_list

Procedure `sp_cursor_list` provides information about all active cursors opened within the current connection to SQL Server.

Main Capabilities:

- Reports on status and characteristics of cursors, such as scrollability, ability to update data, and other important attributes.
- Applied for diagnosing performance problems, monitoring cursor activity, and understanding database connection states.

Input Parameters:

- `[@cursor_return = ] cursor_return OUTPUT`: Cursor variable where procedure results will be placed. Required.
- `[@cursor_scope = ] cursor_scope`: Defines cursor reporting scope:
- 1: Local cursors only.
- 2: Global cursors only.
- 3: All cursors (local + global).

Output Parameters:

- Results returned as Transact-SQL cursor containing information about each opened

cursor:

- Cursor attributes (type, scroll direction, record count, etc.)
- Cursor characteristics (status, update capability, etc.)

Typical Usage Scenarios:

- Diagnosing slow operations caused by inefficient cursor usage.
- Monitoring server load and cursor activity.
- Investigating stuck cursors and resolving related errors.

Query Examples:

```
-- Create cursor variable for receiving sp_cursor_list results
DECLARE @Report AS CURSOR;
-- Execute sp_cursor_list, collecting information about global cursors
EXECUTE sp_cursor_list
@cursor_return = @Report OUTPUT,
@cursor_scope = 2;
-- Read results
FETCH NEXT FROM @Report;
WHILE (@@FETCH_STATUS <> -1)
BEGIN
    FETCH NEXT FROM @Report;
END
-- Close and release resources
CLOSE @Report;
DEALLOCATE @Report;
```

sp_databases

Procedure `sp_databases` returns list of all databases installed on current SQL Server instance.

Main Capabilities:

- Provides information about each database, including its name and size.
- Convenient for quick overview of available databases and assessing their memory consumption.

Input Parameters:

- This procedure does not accept any input parameters.

Output Parameters:

- Result is a table with two main fields:
- DATABASE_NAME: Database name.
- DATABASE_SIZE: Database size in kilobytes.

Typical Usage Scenarios:

- Awareness of list of available databases on server.
- Assessing database sizes for resource planning and performance optimization.

Query Examples:

```
-- Get list of all databases on server  
EXECUTE sp_databases;
```

sp_datatype_info

Procedure sp_datatype_info returns detailed information about data types supported by current database management system (DBMS).

Main Capabilities:

- Provides information about capabilities and characteristics of data types available in SQL Server.
- Allows developers to assess range of capabilities for data processing and query building.

Input Parameters:

- [@data_type =] data_type: Numeric code of data type. If not specified, returns information about all data types.
- [@ODBCVer =] ODBCVer: ODBC standard version used for data type compliance. Possible values: 2 (default) or 3.

Output Parameters:

- Returned table contains information about each data type, including:
- Data type name (TYPE_NAME)
- Case sensitivity indicators (CASE_SENSITIVE)

- Nullability (NULLABLE)
- Minimum and maximum scales (MINIMUM_SCALE, MAXIMUM_SCALE)

Typical Usage Scenarios:

- Designing data models and determining field characteristics.
- Documenting database architecture and supporting developers.

Query Examples:

```
-- Get information about data types nvarchar and sysname  
EXECUTE sp_datatype_info -8;
```

sp_defaultdb

Stored procedure sp_defaultdb is designed to change default database for SQL Server user.

Main Capabilities:

- Assigns default database for specified user or group.
- Improves user convenience when working with database, setting starting connection point.

Input Parameters:

- [@loginame =] N'loginame': Name of user or group for which default database is being set.
- [@defdb =] N'defdb': Name of database that will become standard for specified user.

Output Parameters:

- Code of successful operation execution.
- Return code: 0 (success) or 1 (failure).

Features:

- Cannot be called within user transaction.
- Requires corresponding ALTER ANY LOGIN permission.

Query Examples:

```
EXECUTE sp_defaultdb 'Victoria', 'AdventureWorks2022';
```

sp_describe_cursor

Procedure `sp_describe_cursor` provides information about global attributes of server cursor, such as scrollability, data update capability, and cursor characteristics themselves.

Main Capabilities:

- Reports details about created cursors, facilitating diagnosis and understanding of cursor behavior.
- Helps identify performance problems and suboptimal cursor configurations.

Input Parameters:

- `[@cursor_return = ] cursor_return OUTPUT`: Cursor variable where procedure results will be returned. Required parameter.
- `[@cursor_source = ] { N'local' | N'global' | N'cursor_source' }`: Specifies cursor source (local, global, or cursor variable). No default value.
- `[@cursor_identity = ] N'cursor_identity'`: Unique name of cursor created by `DECLARE CURSOR` statement. Required parameter.

Output Parameters:

- Data returned as Transact-SQL cursor containing information about the cursor itself:
- Cursor attributes (scroll, update, etc.)
- Cursor characteristics (mode, behavior, etc.)

Typical Usage Scenarios:

- Diagnosing malfunctions related to cursor behavior.
- Improving query performance and efficient cursor usage.

Query Examples:

```
DECLARE @Report AS CURSOR;  
EXECUTE sp_describe_cursor  
@cursor_return = @Report OUTPUT,  
@cursor_source = N'global',  
@cursor_identity = N'abc';  
FETCH NEXT FROM @Report;  
WHILE (@@FETCH_STATUS <> -1)  
BEGIN  
    FETCH NEXT FROM @Report;  
END  
CLOSE @Report;  
DEALLOCATE @Report;
```

sp_dropextendedproperty

Procedure `sp_dropextendedproperty` removes existing extended property from SQL Server database object.

Main Capabilities:

- Allows removing additional metadata previously set for database objects (tables, columns, schemas, etc.).
- Useful for cleaning up outdated information and maintaining order in database structure.

Input Parameters:

- `[@name = ] N'name'`: Name of property to be removed. Required parameter.
- `[@level0type = ] 'level0type'`: Type of top-level object (e.g., SCHEMA, TABLE, COLUMN, etc.). Optional parameter.
- `[@level0name = ] N'level0name'`: Name of top-level object. Optional parameter.
- `[@level1type = ] 'level1type'`: Type of next level object. Optional parameter.
- `[@level1name = ] N'level1name'`: Name of next level object. Optional parameter.
- `[@level2type = ] 'level2type'`: Type of last level object. Optional parameter.
- `[@level2name = ] N'level2name'`: Name of last level object. Optional parameter.

Output Parameters:

- Returns execution result:

- 0: successfully removed property.
- 1: removal error.

Typical Usage Scenarios:

- Removing unnecessary or outdated metadata attached to various database objects.
- Maintaining cleanliness and transparency of database structure.

Query Examples:

```
CREATE TABLE T1 (  
    id INT,  
    name CHAR(20)  
);  
EXECUTE sp_addextendedproperty  
@name = 'caption',  
@value = 'Employee ID',  
@level0type = 'SCHEMA',  
@level0name = N'dbo',  
@level1type = 'TABLE',  
@level1name = N'T1',  
@level2type = 'COLUMN',  
@level2name = N'id';  
EXECUTE sp_dropextendedproperty  
@name = 'caption',  
@level0type = 'SCHEMA',  
@level0name = N'dbo',  
@level1type = 'TABLE',  
@level1name = N'T1',  
@level2type = 'COLUMN',  
@level2name = N'id';  
DROP TABLE T1;
```

sp_droplinkedserverlogin

Procedure `sp_droplinkedserverlogin` removes mapping between login name on local server and login name on linked server.

Main Capabilities:

- Removes previously established mapping for linked server, allowing elimination of unwanted or erroneous mappings.
- Restores default mapping if remote user is no longer needed.

Input Parameters:

- [**@rmtsrvname** =] N'rmtsrvname': Name of linked server for which mapping is being removed. Required parameter.
- [**@locallogin** =] N'locallogin': Name of local login whose account on linked server needs to be deleted. If set to NULL, removes default mapping for all local users.

Output Parameters:

- Execution result:
- 0: successfully removed mapping.
- 1: removal failure.

Typical Usage Scenarios:

- Eliminating incorrect or unused mappings interfering with normal operation.
- Bringing database security system in order after infrastructure changes.

Query Examples:

```
-- Remove mapping of user Mary on linked server Accounts  
EXECUTE sp_droplinkedserverlogin 'Accounts', 'Mary';
```

```
-- Remove default mapping for all users on linked server Accounts  
EXECUTE sp_droplinkedserverlogin 'Accounts', NULL;
```

sp_droprole

Procedure **sp_droprole** removes existing database role from current database.

Main Capabilities:

- Removes specified database role, freeing space and cleaning security system.
- Ease of managing security structures and administrative resources.

Input Parameters:

- [**@rolename** =] N'rolename': Name of database role to be deleted. Required parameter.

Output Parameters:

- Execution result:
- 0: successfully removed role.
- 1: removal error.

Typical Usage Scenarios:

- Removing unnecessary or outdated database roles to improve clarity and maintenance simplicity.
- Cleaning database security infrastructure after reorganization.

Query Examples:

```
-- Remove role Sales  
EXEC sp_droprole 'Sales';
```

sp_droprolemember

Procedure sp_droprolemember removes user or other member from database role.

Main Capabilities:

- Allows excluding users or groups from database roles, eliminating redundant or unreliable associations.
- Promotes security management and data protection.

Input Parameters:

- [@rolename =] N'rolename': Name of database role from which member is being removed. Required parameter.
- [@membername =] N'membername': Name of user or group being removed from role. Required parameter.

Output Parameters:

- Execution result:
- 0: successfully executed exclusion from role.
- 1: process error.

Typical Usage Scenarios:

- Excluding employees who left company from corresponding roles.
- Enhancing database security by eliminating unnecessary role membership.

Query Examples:

```
-- Exclude user JonB from role Sales  
EXECUTE sp_droprolemember 'Sales', 'JonB';
```

sp_dropserver

Procedure sp_dropserver removes server from list of known remote or linked servers on local SQL Server instance.

Main Capabilities:

- Allows getting rid of unnecessary connections to remote servers, preventing possible conflicts and errors.
- Maintains cleanliness of SQL Server infrastructure and improves control over network connections.

Input Parameters:

- [@server =] N'server': Name of server to be removed. Required parameter.
- [@droplogins =] 'droplogins': Flag indicating need for simultaneous removal of associated logins. Default: NULL.

Output Parameters:

- Execution result:
- 0: successfully removed server.
- 1: removal error.

Typical Usage Scenarios:

- Removing references to old or incorrectly configured remote servers.
- Maintaining clean and efficient SQL Server infrastructure.

Query Examples:

```
-- Remove remote server ACCOUNTS and all associated logins  
EXECUTE sp_dropserver 'ACCOUNTS', 'droplogins';
```

sp_dropuser

Stored procedure sp_dropuser removes user from current database.

Main Capabilities:

- Removes database user account.
- Eliminates unnecessary load on security system.

Input Parameters:

- [@name_in_db =] N' name_in_db': Name of user to be deleted.

Output Parameters:

- Code of successful operation execution.
- Return code: 0 (success) or 1 (failure).

Features:

- Before deleting user, ensure they do not own database objects.
- Requires ALTER ANY USER permissions.

Query Examples:

```
EXECUTE sp_dropuser 'Albert';
```

sp_enumoledb_providers

Procedure sp_enumoledb_providers returns list of registered OLE DB providers available in current SQL Server environment.

Main Capabilities:

- Provides comprehensive information about OLE DB providers needed for integration with external data sources.
- Useful for selecting appropriate provider when configuring connection to third-party systems.

Input Parameters:

- Procedure requires no input parameters.

Output Parameters:

- Returns table with list of registered OLE DB providers, including:
- Provider name (NAME)
- Version (VERSION)
- Information about specialized function support.

Typical Usage Scenarios:

- Planning integration processes with external data sources.
- Diagnosing problems related to missing required OLE DB provider.

Query Examples:

```
-- Get list of registered OLE DB providers  
EXEC sp_enum_oledb_providers;
```

sp_fkeys

Procedure sp_fkeys returns information about logical foreign keys active in current database.

Main Capabilities:

- Provides complete view of relationships between tables through foreign keys.
- Includes information even about foreign keys that are temporarily disabled.

Input Parameters:

- [@pktable_name =] N'pktable_name': Name of table with primary key. Required parameter.
- [@pktable_owner =] N'pktable_owner': Name of table with primary key owner. Optional parameter.
- [@pktable_qualifier =] N'pktable_qualifier': Name of table with primary key qualifier (e.g., database name). Optional parameter.

- [**@fktable_name** =] N'fktable_name': Name of table with foreign key. Optional parameter.
- [**@fktable_owner** =] N'fktable_owner': Name of table with foreign key owner. Optional parameter.
- [**@fktable_qualifier** =] N'fktable_qualifier': Name of table with foreign key qualifier (e.g., database name). Optional parameter.

Output Parameters:

- Table with results, including:
- Names of tables with primary and foreign keys.
- Cascade update and delete rules.
- Sequence of key columns.

Typical Usage Scenarios:

- Understanding table dependencies on each other.
- Generating diagrams and visualizing relationships between tables.

Query Examples:

```
-- Get information about foreign keys of table HumanResources.Department
EXECUTE sp_fkeys
@pktable_name = N'Department',
@pktable_owner = N'HumanResources';
```

sp_getapplock

Procedure **sp_getapplock** sets lock on application resource, allowing control of concurrent access to this resource.

Main Capabilities:

- Sets lock on specific resource, preventing access conflict between concurrent sessions or transactions.
- Guarantees sequential access to critically important data elements or business process logic.

Input Parameters:

- [**@Resource =] N'Resource'**: Name of resource on which lock is applied. Maximum 255 characters.
- [**@LockMode =] 'LockMode'**: Lock mode: Shared, Update, IntentShared, IntentExclusive, Exclusive.
- [**@LockOwner =] 'LockOwner'**: Lock owner (Transaction or Session). Default: Transaction.
- [**@LockTimeout =] LockTimeout**: Lock wait time in milliseconds. Default: infinite.
- [**@DbPrincipal =] N'DbPrincipal'**: Database principal having rights to locked object. Default: public.

Output Parameters:

- Execution result:
- 0: lock successfully set.
- -1: timeout expired.
- -2: lock request canceled.
- -3: deadlock situation occurred.
- Other negative values indicate various errors.

Typical Usage Scenarios:

- Controlling concurrent access to shared resources (e.g., form or process).
- Implementing mechanism to prevent conflicts between users or clients.

Query Examples:

```
-- Set shared lock on resource Form1
EXECUTE sp_getapplock
@Resource = 'Form1',
@LockMode = 'Shared';
```

sp_help

Stored procedure `sp_help` provides detailed information about database objects such as tables, views, data types, and stored procedures.

Main Capabilities:

- Shows structure and properties of database objects.
- Helps developers and administrators understand database schema.

Input Parameters:

- [**@objname =] N'objname'**: Name of database object (table, view, data type, etc.). If parameter not specified, procedure returns list of all objects in current database.

Output Parameters:

- Information about object structure, including name, owner, type, column sizes, indexes, constraints, and other characteristics.
- Return type: No specific type; result presented as result set.

Features:

- Works only with objects of current database.
- Requires at least one permission to requested object.

Query Examples:

```
--View entire database structure  
EXEC sp_help;
```

sp_helpdb

Procedure **sp_helpdb** returns information about database or all databases on current SQL Server instance.

Main Capabilities:

- Provides detailed information about size, owner, creation date, and database parameters.
- Used for general monitoring and analysis of database infrastructure.

Input Parameters:

- [**@dbname =] N'dbname'**: Name of database for which information is needed. If not specified, returns information about all databases.

Output Parameters:

- Returns table with data about each database, including:
- Database name (name)
- Total size (db_size)
- Owner (owner)
- Creation date (created)
- Current parameters (status)
- Compatibility level (compatibility_level)

Typical Usage Scenarios:

- Quick assessment of database sizes and states.
- Collecting general information for performance monitoring and maintenance planning. *Query Examples:*

```
-- Get information about database AdventureWorks2022  
EXECUTE sp_helpdb N'AdventureWorks2022';
```

sp_helpdbfixedrole

Procedure `sp_helpdbfixedrole` returns list of fixed database roles and their description.

Main Capabilities:

- Provides information about built-in database roles, such as `db_owner`, `db_accessadmin`, `db_securityadmin`, and others.
- These roles provide convenient way to manage access rights at database level.

Input Parameters:

- `[@rolename = ] N'rolename'`: Name of fixed database role. If not specified, returns information about all fixed roles.

Output Parameters:

- Table with results, including two columns:
- Fixed role name (DbFixedRole)
- Role description (Description)

Typical Usage Scenarios:

- Administrators and developers frequently refer to this list for better understanding of access levels and privileges provided by fixed roles.
- Using fixed roles simplifies access rights management and separation of administrative duties. *Query Examples:*

```
-- Get list of all fixed database roles  
EXECUTE sp_helpdbfixedrole;
```

sp_helplinkedsrvlogin

Procedure sp_helplinkedsrvlogin provides information about mappings between login names on local server and login names on linked server.

Main Capabilities:

- Returns information about existing mappings for distributed queries and remote stored procedures.
- Used for analysis and diagnosis of linked database security infrastructure.

Input Parameters:

- [*@rmtsrvname* =] N'rmtsrvname': Name of linked server. If not specified, returns information about all mappings on all linked servers.
- [*@locallogin* =] N'locallogin': Name of local login for which mapping is being searched. If not specified, returns all mappings for specified linked server.

Output Parameters:

- Table with results, including:
 - Linked server name (Linked Server)
 - Local login name (Local Login)
 - Self-mapping indicator (Is Self Mapping)
 - Remote login name (Remote Login)

Typical Usage Scenarios:

- Understanding how local server users interact with remote servers.

- Diagnosing security and access problems to linked servers.

Query Examples:

```
-- Get information about all mappings on all linked servers  
EXECUTE sp_helplinkedsrvlogin;
```

sp_helprole

Procedure sp_helprole returns information about roles existing in current database.

Main Capabilities:

- Provides list of all roles available in current database, including their identifiers and application role membership indicator.
- Used for monitoring and managing database security structure.

Input Parameters:

- [*@rolename* =] N'*rolename*': Name of role for which information is needed. If not specified, returns information about all roles.

Output Parameters:

- Table with results, including:
 - Role name (RoleName)
 - Role identifier (RoleId)
 - Application role membership indicator (IsAppRole)

Typical Usage Scenarios:

- Analyzing database security structure and identifying risks.
- Maintaining order and transparency when managing database access.

Query Examples:

```
-- Get information about all roles in current database  
EXECUTE sp_helprole;
```

sp_helprolemember

Procedure `sp_helprolemember` returns information about direct members of specified role in current database.

Main Capabilities:

- Presents list of users and roles that are direct participants of role.
- Used for diagnosis and management of database security.

Input Parameters:

- `[@rolename = ] N'rolename'`: Name of role whose members need to be listed. If not specified, returns information about all roles having at least one direct member.

Output Parameters:

- Table with results, including:
- Database role name (DbRole)
- Member name (MemberName)
- Member security identifier (MemberSID)

Typical Usage Scenarios:

- Analyzing role composition to identify potential security threats.
- Managing database security infrastructure and monitoring role membership.

Query Examples:

```
-- Get information about members of role Sales  
EXECUTE sp_helprolemember 'Sales';
```

sp_helpsrvrolemember

Procedure `sp_helpsrvrolemember` returns information about members of fixed server roles in SQL Server.

Main Capabilities:

- Provides list of users and roles that are members of fixed server roles.
- Used for diagnosis and management of server-level security infrastructure.

Input Parameters:

- [*@srvrolename* =] N'srvrolename': Name of fixed server role. If not specified, returns information about all fixed roles.

Output Parameters:

- Table with results, including:
- Server role name (ServerRole)
- Member name (MemberName)
- Member security identifier (MemberSID)

Typical Usage Scenarios:

- Administrators regularly use this procedure to ascertain composition of fixed server roles.
- Maintaining transparency and integrity of SQL Server security system.

Query Examples:

```
-- Get information about members of role sysadmin  
EXECUTE sp_helsrvrolemember 'sysadmin';
```

sp_helpuser

Procedure `sp_helpuser` returns information about users and roles present in current database.

Main Capabilities:

- Provides overview of all users and their roles in current database.
- Helps manage security infrastructure and quickly diagnose potential threats.

Input Parameters:

- [*@name_in_db* =] N'name_in_db': Name of user or role for which information is needed. If not specified, returns information about all users and roles.

Output Parameters:

- Table with results, including:
- User name (UserName)

- Assigned roles (RoleName)
- Login account (LoginName)
- Default base database (DefDBName)
- Default schema (DefSchemaName)

Typical Usage Scenarios:

- General diagnosis and inventory of database users and roles.
- Maintaining secure and transparent database infrastructure.

Query Examples:

```
-- Get information about all users of current database  
EXECUTE sp_helpuser;
```

sp_indexes_rowset

Stored procedure sp_indexes_rowset retrieves information about indexes for given object in database.

Main Capabilities:

- Provides detailed information about presence and characteristics of table indexes.
- Used for analyzing query performance and diagnosing performance problems.

Input Parameters:

- [@table_server =] N'table_server': Server where object is located.
- [@table_catalog =] N'table_catalog': Catalog (database) containing object.
- [@table_schema =] N'table_schema': Schema to which object belongs.
- [@table_name =] N'table_name': Name of object (table or view) for which index information is needed.

Output Parameters:

- Set of rows, each describing a separate index.
- Contains information about index type, unique columns, clustering indicators, and much more.

- Return code: 0 (success) or 1 (failure).

Features:

- Applied primarily for diagnostic purposes and query optimization.
- Best used together with query profiling tool and performance monitoring.

Query Examples:

```
Get information about indexes of table Orders
EXEC sp_indexes_rowset
    @table_server = 'localhost',
    @table_catalog = 'SalesDatabase',
    @table_schema = 'dbo',
    @table_name = 'Orders';
```

sp_linkedservers

Procedure sp_linkedservers returns list of all linked servers configured on current SQL Server instance.

Main Capabilities:

- Provides complete list of linked servers, including their network names, provider names, product names, and other characteristics.
- Used for diagnosis and monitoring of linked server infrastructure.

Input Parameters:

- Procedure accepts no input parameters.

Output Parameters:

- Table with results, including:
- Linked server name (SRV_NAME)
- OLE DB provider (SRV_PROVIDERNAME)
- Linked server product (SRV_PRODUCT)
- Data source (SRV_DATASOURCE)
- Provider string (SRV_PROVIDERSTRING)
- Location (SRV_LOCATION)

- Catalog (SRV_CAT)

Typical Usage Scenarios:

- Inventory of all linked servers for monitoring availability and health.
- Solving access problems to linked servers and configuring correct infrastructure.

Query Examples:

```
-- Get list of all linked servers  
EXECUTE sp_linkedservers;
```

sp_oledb_ro_username

Procedure sp_oledb_ro_username returns name of user last used to connect to server using OLE DB provider.

Main Capabilities:

- Provides information about last used user name for identifying recent connection attempts.
- Can be applied for diagnosing access or security problems.

Input Parameters:

- Procedure accepts no input parameters.

Output Parameters:

- Returns user name last used for connection via OLE DB.

Typical Usage Scenarios:

- Monitoring connection activity via OLE DB.
- Diagnosing access problems and user identification.

Query Examples:

```
-- Get user name last used for connection via OLE DB  
EXEC sp_oledb_ro_username;
```


sp_password

Stored procedure sp_password allows changing password for SQL Server login.

Main Capabilities:

- Changes password of registered user.
- Increases level of authentication system protection.

Input Parameters:

- [@old =] N'old': Old password (if not using named parameter assignment).
- [@new =] N'new': New password.
- [@loginame =] N'loginame': User name for which password is being changed.

Output Parameters:

- Code of successful operation execution.
- Return code: 0 (success) or 1 (failure).

Features:

- User must choose strong passwords.
- Password cannot be changed within transaction.

Query Examples:

```
Setting new password  
ALTER LOGIN Victoria WITH PASSWORD = '<password>';
```

sp_pkeys

Procedure sp_pkeys returns information about primary keys of table in current database.

Main Capabilities:

- Provides list of columns constituting primary key of specified table.
- Used for understanding database structure and mechanisms of unique record identification.

Input Parameters:

- `[@table_name = ] N'table_name'`: Name of table for which primary key information is needed. Required parameter.
- `[@table_owner = ] N'table_owner'`: Name of table owner. Optional parameter.

Output Parameters:

- Table with results, including:
- Table name (TABLE_NAME)
- Column name (COLUMN_NAME)
- Column ordinal number in primary key composition (KEY_SEQ)
- Primary key name (PK_NAME)

Typical Usage Scenarios:

- Understanding database structure and ensuring unique record identification.
- Analyzing impact of primary keys on data integrity and query efficiency.

Query Examples:

```
-- Get information about primary key of table HumanResources.Department
EXEC sp_pkeys
@table_name = N'Department',
@table_owner = N'HumanResources';
```

sp_prepare

Stored procedure `sp_prepare` prepares parameterized Transact-SQL query and returns unique identifier (handle) for subsequent execution.

Main Capabilities:

- Pre-compiling query for improving efficiency of repeated execution.
- Returns handle that can be reused for efficient execution of similar queries.

Input Parameters:

- `handle`: Variable where prepared statement handle will be saved (data type: int).
- `params`: List of query parameters in format `@param_name datatype`.

- stmt: Transact-SQL command itself that needs to be prepared (data type: ntext, nchar or nvarchar).
- options: Optional parameter for setting additional preparation settings.

Output Parameters:

- Unique handle of prepared query that can be used for further execution via sp_execute procedure.
- Return code: 0 (success) or 1 (failure).

Features:

- Prepares execution plan for reuse, reducing recompilation costs.
- Preparation plan remains active until manually removed via sp_unprepare procedure.

Query Examples:

```
Prepare and execute simple query
DECLARE @handle AS INT;
EXECUTE sp_prepare
@handle OUTPUT, N'
@P1 NVARCHAR(128),
@P2 NVARCHAR(100)',
N'SELECT database_id, name FROM sys.databases WHERE name=@P1 AND state_desc = @P2';
EXECUTE sp_execute
@handle,
N'tempdb', N'ONLINE';
EXECUTE sp_unprepare @handle;
```

sp_primary_keys_rowset

Stored procedure sp_primary_keys_rowset retrieves information about primary keys for given object in database.

Main Capabilities:

- Provides detailed information about columns included in primary key of table.
- Used for analyzing data structure and designing database schemas.

Input Parameters:

- [`@table_server = ] N'table_server'`: Server where table is located.
- [`@table_catalog = ] N'table_catalog'`: Catalog (database) containing table.
- [`@table_schema = ] N'table_schema'`: Schema to which table belongs.
- [`@table_name = ] N'table_name'`: Name of table for which primary key information is needed.
- [`@table_qualifier = ] N'table_qualifier'`: Name of table qualifier (e.g., database name). Optional parameter.

Output Parameters:

- Set of rows, each describing a separate column of primary key.
- Contains information about column name, column order in key, and column data type.
- Return code: 0 (success) or 1 (failure).

Features:

- Usually used for internal analysis of database structure.
- Best applied together with other database structure analysis tools.

Query Examples:

```
A. Get information about primary key of table Employees
EXEC sp_primary_keys_rowset
    @table_server = 'localhost',
    @table_catalog = 'HRSystem',
    @table_schema = 'dbo',
    @table_name = 'Employees';
```

sp_procedure_params_100_managed

Procedure `sp_procedure_params_100_managed` provides information about parameters of stored procedures and functions located in managed environment (e.g., SQL Azure or Azure SQL Managed Instance).

Main Capabilities:

- Returns list of parameters for specific procedure or function.
- Allows analyzing interface of procedure or function and understanding which

parameters are accepted and how they are processed.

Input Parameters:

- [@procedure_schema =] N'procedure_schema': Name of schema where procedure or function is located.
- [@object_name =] N'object_name': Name of procedure or function itself.

Output Parameters:

- Table with results, including:
- Parameter name (parameter_name)
- Data type (data_type)
- Direction (input/output) (parameter_mode)
- Required parameter indicator (is_required)

Typical Usage Scenarios:

- Analyzing interfaces of stored procedures and functions for software design and development.
- Determining parameters for calling stored procedures and functions.

Query Examples:

```
-- Get information about parameters of procedure usp_GetOrdersByCustomer  
EXEC sp_procedure_params_100_managed N'dbo', N'usp_GetOrdersByCustomer';
```

sp_releaseapplock

Procedure sp_releaseapplock releases lock from application resource previously set by sp_getapplock procedure.

Main Capabilities:

- Releases previously set resource lock, allowing other sessions to access resource.
- Helps prevent long-term locks and improve database work efficiency.

Input Parameters:

- [@Resource =] N'Resource': Name of resource from which lock is being released.

Required parameter.

- [**@LockOwner** =] 'LockOwner': Lock owner (Transaction or Session). Default: Transaction.
- [**@DbPrincipal** =] N'DbPrincipal': Database principal having rights to locked object. Default: public.

Output Parameters:

- Execution result:
- 0: lock successfully released.
- -999: error releasing lock or other problems. *Typical Usage Scenarios:*
- Need to release lock previously applied by `sp_getapplock` procedure to allow other sessions to access resource.
- Need to organize resource release for improving database performance.

Query Examples:

```
BEGIN TRANSACTION;  
DECLARE @result INT;  
EXEC @result = sp_getapplock  
    @DbPrincipal = 'dbo',  
    @Resource = 'test_2',  
    @LockMode = 'Exclusive';  
EXECUTE sp_releaseapplock  
    @DbPrincipal = 'dbo',  
    @Resource = 'test_2';  
COMMIT TRANSACTION;
```

sp_renamedb

Procedure `sp_renamedb` allows changing name of existing database.

Main Capabilities:

- Changes current database name to new name.
- Used for renaming databases, especially when changing their purpose or restructuring infrastructure.

Input Parameters:

- [**@dbname =] N'dbname'**: Current database name to be changed. Required parameter.
- [**@newname =] N'newname'**: New database name. Required parameter.

Output Parameters:

- Execution result:
- 0: successfully renamed.
- Non-zero number: error occurred.

Typical Usage Scenarios:

- Renaming database during infrastructure changes or transition to new naming standards.
- Maintaining consistency and convenience of database operation.

Query Examples:

```
-- Rename database Accounting to Financial  
EXECUTE sp_renamedb N'Accounting', N'Financial';
```

sp_serveroption

Procedure `sp_serveroption` allows setting and changing parameters for remote and linked servers.

Main Capabilities:

- Configures communication and interaction parameters with remote and linked servers.
- Allows flexible configuration of interaction with external data sources, affecting speed and reliability of distributed queries.

Input Parameters:

- [**@server =] N'server'**: Name of remote or linked server for which parameters are being set. Required parameter.
- [**@optname =] 'optname'**: Name of parameter to be changed. For example, collation compatible, connect timeout, rpc, etc. Required parameter.

- [*@optvalue* =] *N'optvalue'*: New parameter value. For example, true, false, on, off, or specific value, e.g., timeout in seconds. Required parameter.

Output Parameters:

- Execution result:
- 0: successfully changed parameter.
- 1: parameter change error.

Typical Usage Scenarios:

- Configuring distributed queries and remote procedure calls.
- Optimizing speed and efficiency of query execution to linked servers.

Query Examples:

```
-- Configure collation compatibility parameter for linked server SEATTLE3  
EXECUTE sp_serveroption N'SEATTLE3', 'collation compatible', N'true';
```

sp_spaceused

Stored procedure `sp_spaceused` displays information about number of rows, reserved and used disk space for database objects.

Main Capabilities:

- Provides detailed picture of database resource distribution.
- Helps assess need for free space and plan maintenance.

Input Parameters:

- [*@objname* =] *N'objname'*: Name of object (table, view, Service Broker queue).
- [*@updateusage* =] *'updateusage'*: Indicator of forced update of space usage statistics.
- [*@mode* =] *'mode'*: Information display mode (local, remote, or full).
- [*@onerresultset* =] *onerresultset*: Choice of single result set return mode.
- [*@include_total_xtp_storage* =] *include_total_xtp_storage*: Includes additional information about checkpoint files for Memory Optimized Data tables.

Output Parameters:

- Detailed space usage statistics.
- Return code: 0 (success) or 1 (failure).

Features:

- Available to all users.
- Statistics update mandatory for large objects.

Query Examples:

```
-- Information about disk space of table  
EXECUTE sp_spaceused N'Purchasing.Vendor';
```

sp_special_columns

Procedure `sp_special_columns` returns information about special columns of table, such as unique row identifiers and automatically updating columns.

Main Capabilities:

- Determines minimum set of columns ensuring unique identification of rows in table.
- Provides information about columns that automatically update when any row changes.

Input Parameters:

- `[@table_name = ] N'table_name'`: Name of table for which special column information is needed. Required parameter.
- `[@table_owner = ] N'table_owner'`: Name of table owner. Optional parameter.
- `[@table_qualifier = ] N'table_qualifier'`: Name of table qualifier (e.g., database name). Optional parameter.
- `[@col_type = ] 'col_type'`: Type of special column: R (optimal row identifier) or V (auto-updating column). Default: R.
- `[@scope = ] 'scope'`: Required scope of row identifier: C (row), T (transaction). Default: T.
- `[@nullable = ] 'nullable'`: Whether columns can accept NULL value: * (no), U (yes).

Default: U.

- [@ODBCVer =] ODBCVer: ODBC version. Default: 2.

Output Parameters:

- Table with results, including:
- Actual scope of row identifier (SCOPE)
- Column name (COLUMN_NAME)
- Data type (DATA_TYPE)
- Column scale and length (PRECISION, LENGTH, SCALE)

Typical Usage Scenarios:

- Understanding structure of unique row identifiers in tables.
- Determining automatically updating columns for improved synchronization and change management.

Query Examples:

```
-- Get information about special column of table HumanResources.Department
EXECUTE sp_special_columns
@table_name = 'Department',
@table_owner = 'HumanResources';
```

sp_sproc_columns

Procedure sp_sproc_columns returns information about columns of stored procedure or user function in current database.

Main Capabilities:

- Provides information about number and types of procedure or function parameters.
- Allows ascertaining structure and functionality of procedures and functions for further work with them.

Input Parameters:

- [@procedure_name =] N'procedure_name': Name of procedure or function for which column information is needed. Required parameter.

- `[@procedure_owner = ] N'procedure_owner'`: Name of procedure or function owner. Optional parameter.
- `[@procedure_qualifier = ] N'procedure_qualifier'`: Name of procedure or function qualifier (e.g., database name). Optional parameter.
- `[@column_name = ] N'column_name'`: Limits result to one column. If not specified, returns all columns. Optional parameter.
- `[@ODBCVer = ] ODBCVer`: ODBC version. Default: 2.
- `[@fUsePattern = ] fUsePattern`: Enables or disables pattern matching for parameters. Default: enabled (1).

Output Parameters:

- Table with results, including:
- Procedure name (PROCEDURE_NAME)
- Column name (COLUMN_NAME)
- Data type (DATA_TYPE)
- Data sizes and format (PRECISION, LENGTH, SCALE)

Typical Usage Scenarios:

- Analyzing structure of stored procedures and functions for documentation or modification.
- Checking number and types of procedure or function parameters before calling them.

Query Examples:

```
-- Get information about columns of procedure usp_GetOrdersByCustomer  
EXEC sp_sproc_columns @procedure_name = N'usp_GetOrdersByCustomer';
```

sp_statistics

Procedure `sp_statistics` returns information about all indexes and statistical information for specified table or indexed view.

Main Capabilities:

- Provides list of all indexes and statistics for table or view.

- Used for analyzing index structure and assessing their usefulness for query optimization.

Input Parameters:

- [`@table_name = ] N'table_name'`: Name of table or view for which information is required. Required parameter.
- [`@table_owner = ] N'table_owner'`: Name of table owner. Optional parameter.
- [`@table_qualifier = ] N'table_qualifier'`: Name of table qualifier (e.g., database name). Optional parameter.
- [`@index_name = ] N'index_name'`: Name of index. If not specified, returns all indexes. Optional parameter.
- [`@is_unique = ] 'is_unique'`: Indicates whether to output only unique indexes. Possible values: Y (yes), empty string (default, all indexes). Optional parameter.
- [`@accuracy = ] 'accuracy'`: Statistics accuracy level. Possible values: E (get exact data), Q (quickly get approximate statistics). Default: Q. Optional parameter.

Output Parameters:

- Table with results, including:
- Table name (TABLE_NAME)
- Index name (INDEX_NAME)
- Index type (TYPE)
- Number of rows or unique values (CARDINALITY)
- Number of pages for storing index (PAGES)

Typical Usage Scenarios:

- Analyzing index structure for optimal query tuning.
- Determining needed indexes for query acceleration.

Query Examples:

```
-- Get information about all indexes of table DimEmployee  
EXECUTE sp_statistics DimEmployee;
```

sp_stored_procedures

Procedure `sp_stored_procedures` returns list of all stored procedures in current database.

Main Capabilities:

- Provides complete list of stored procedures, including their owners and schemas.
- Allows quickly checking procedure presence and familiarizing with their composition.

Input Parameters:

- `[@sp_name = ] N'sp_name'`: Name of procedure for which information is needed. If not specified, returns all procedures.
- `[@sp_owner = ] N'sp_owner'`: Name of procedure owner. Optional parameter.
- `[@sp_qualifier = ] N'sp_qualifier'`: Name of procedure qualifier (e.g., database name). Optional parameter.
- `[@fUsePattern = ] fUsePattern`: Enables or disables interpretation of special characters (%) and (_) in filtering. Default: enabled (1).

Output Parameters:

- Table with results, including:
- Procedure name (PROCEDURE_NAME)
- Owner name (PROCEDURE_OWNER)
- Other reserved fields (NUM_INPUT_PARAMS, NUM_OUTPUT_PARAMS, etc.)

Typical Usage Scenarios:

- Obtaining complete list of stored procedures for analysis and documentation.
- Searching for specific procedure for further work or editing.

Query Examples:

```
-- Get list of all stored procedures in current database  
EXECUTE sp_stored_procedures;
```

sp_table_privileges

Procedure `sp_table_privileges` returns list of permissions (access rights) assigned to

specific tables in current database.

Main Capabilities:

- Provides information about table access rights, such as INSERT, DELETE, UPDATE, SELECT, REFERENCES.
- Used for analysis and diagnosis of table access rights situation.

Input Parameters:

- [`@table_name = ] N'table_name'`: Name of table for which access rights information is needed. If not specified, returns access rights for all tables.
- [`@table_owner = ] N'table_owner'`: Name of table owner. Optional parameter.
- [`@table_qualifier = ] N'table_qualifier'`: Name of table qualifier (e.g., database name). Optional parameter.
- [`@fUsePattern = ] fUsePattern`: Flag for enabling or disabling replacement pattern for filter. Default: enabled (1).

Output Parameters:

- Table with results, including:
- Table name (TABLE_NAME)
- Table owner (TABLE_OWNER)
- Permission type (PRIVILEGE)
- User who received permission (GRANTEE)
- Whether user has right to pass this permission further (IS_GRANTABLE)

Typical Usage Scenarios:

- Diagnosing situations with insufficient access level to tables.
- Setting tasks for improving database security and protecting confidential data.

Query Examples:

```
-- Get information about access rights to all tables starting with Contact
EXECUTE sp_table_privileges @table_name = 'Contact%';
```

sp_tablecollations_100

Procedure `sp_tablecollations_100` returns information about collation rules for columns of specified table.

Main Capabilities:

- Provides information about data sorting and comparison in table columns.
- Used for analyzing data compatibility and detecting encoding discrepancies.

Input Parameters:

- `[@table_name = ] N'table_name'`: Name of table for which column collation information is needed. Required parameter.
- `[@table_owner = ] N'table_owner'`: Name of table owner. Optional parameter.
- `[@table_qualifier = ] N'table_qualifier'`: Name of table qualifier (e.g., database name). Optional parameter.

Output Parameters:

- Table with results, including:
- Table name (TABLE_NAME)
- Column name (COLUMN_NAME)
- Collation rule (COLLATION_NAME)

Typical Usage Scenarios:

- Diagnosing data compatibility problems and collation rule discrepancies.
- Maintaining unified encoding policy in database.

Query Examples:

```
-- Get information about collation rules for table Employees  
EXEC sp_tablecollations_100 N'Employees';
```

sp_tables

Procedure `sp_tables` returns list of all tables, views, and system tables available for queries in current database.

Main Capabilities:

- Provides complete list of objects that can be accessed with queries in current database.
- Useful for analyzing database structure and searching for necessary objects.

Input Parameters:

- [`@table_name = ] N'table_name'`: Name of table or view for which information is needed. If not specified, returns information about all objects.
- [`@table_owner = ] N'table_owner'`: Name of table or view owner. Optional parameter.
- [`@table_qualifier = ] N'table_qualifier'`: Name of table qualifier (e.g., database name). Optional parameter.
- [`@table_type = ] 'table_type'`: Type of objects to return (e.g., TABLE, VIEW, SYSTEMTABLE). Optional parameter.
- [`@fUsePattern = ] fUsePattern`: Enables or disables interpretation of special characters (`%`, `_`, `[`, `]`) in filtering. Default: enabled (1).

Output Parameters:

- Table with results, including:
- Table name (TABLE_NAME)
- Owner name (TABLE_OWNER)
- Object type (TABLE_TYPE)

Typical Usage Scenarios:

- Analyzing database structure and searching for specific tables or views.
- Creating complete map of database objects for further use. *Query Examples:*

```
-- Get list of all objects available for queries  
EXECUTE sp_tables;
```

sp_unbindefault

Stored procedure `sp_unbindefault` serves to remove bound default values from columns or data type aliases.

Main Capabilities:

- Removes set default value for column or data type.
- Allows selectively preserving existing default values when unbinding for future use.

Input Parameters:

- [**@objname** =] N'objname': Name of object (column or data type) from which default value needs to be removed.
- [**@futureonly** =] 'futureonly': Optional parameter indicating that unbinding applies only to future objects.

Output Parameters:

- Informational message about operation results.
- Return code: 0 (success) or 1 (failure).

Features:

- After unbinding, columns that had their own default value remain unchanged.
- Procedure requires appropriate access rights for execution.

Query Examples:

```
A. Removing default value from column  
EXECUTE sp_unbindefault 'employees.hiredate';
```

sp_unprepare

Stored procedure **sp_unprepare** discards previously created query execution plan prepared by **sp_prepare** procedure.

Main Capabilities:

- Releases resources occupied by saved execution plan.
- Optimizes performance and reduces RAM consumption.

Input Parameters:

- **handle**: Identifier of execution plan obtained from **sp_prepare** procedure. Data type: int.

Output Parameters:

- Command executed successfully, no returned data.
- Return code: 0 (success) or 1 (failure).

Features:

- Executed by client via transmission of special TDS (Tabular Data Stream) packet.
- Strict parameter order must be followed, otherwise failure possible.

Query Examples:

```
Prepare, execute and release simple query
DECLARE @P1 AS INT;
EXECUTE sp_prepare
@P1 OUTPUT, N'@P1 NVARCHAR(128), @P2 NVARCHAR(100)',
N'SELECT database_id, name FROM sys.databases WHERE name = @P1 AND state_desc = @P2';
EXECUTE sp_execute @P1, N'tempdb', N'ONLINE';
EXECUTE sp_unprepare @P1;
```

sp_updateextendedproperty

Procedure sp_updateextendedproperty updates value of existing extended property in SQL Server database.

Main Capabilities:

- Allows changing metadata attached to database objects (tables, columns, functions, etc.).
- Used for maintaining current and accurate information about database structure.

Input Parameters:

- [**@name =]** N'name': Name of property whose value needs to be updated. Required parameter.
- [**@value =]** value: New property value. Required parameter.
- [**@level0type =]** 'level0type': Type of level 0 object (e.g., SCHEMA, TABLE, COLUMN, etc.). Optional parameter.
- [**@level0name =]** N'level0name': Name of level 0 object. Optional parameter.
- [**@level1type =]** 'level1type': Type of level 1 object. Optional parameter.

- [**@level1name** =] N'level1name': Name of level 1 object. Optional parameter.
- [**@level2type** =] 'level2type': Type of level 2 object. Optional parameter.
- [**@level2name** =] N'level2name': Name of level 2 object. Optional parameter.

Output Parameters:

- Execution result:
- 0: successfully updated property value.
- 1: update error.

Typical Usage Scenarios:

- Maintaining documentation and explanations for database objects.
- Updating metadata for bringing clarity to database structure.

Query Examples:

```
CREATE TABLE T1 (id INT, name CHAR(20));  
EXECUTE sp_addextendedproperty  
@name = N'Caption',  
@value = N'Employee ID',  
@level0type = 'SCHEMA',  
@level0name = N'dbo',  
@level1type = 'TABLE',  
@level1name = N'T1',  
@level2type = 'COLUMN',  
@level2name = N'id';  
-- Update extended property  
EXECUTE sp_updateextendedproperty  
@name = N'Caption',  
@value = 'Employee ID must be unique.',
```

sp_updatestats

Procedure **sp_updatestats** performs UPDATE STATISTICS operation for all user and internal tables in current database.

Main Capabilities:

- Updates statistics for all tables in database, helping optimize query performance.
- Used for regularly maintaining currency of statistics in database.

Input Parameters:

- [**@resample =**] 'resample': Flag indicating procedure should use RESAMPLE option when updating statistics. If not specified, uses standard sampling algorithm.

Output Parameters:

- Execution result:
- 0: successfully updated statistics.
- 1: update error. *Typical Usage Scenarios:*
- Periodically updating statistics for maintaining high query performance.
- Maintaining optimal query system operation after significant data changes.

Query Examples:

```
-- Update statistics for all tables in database AdventureWorks2022  
EXECUTE sp_updatestats;
```

sp_who

Procedure sp_who provides information about current users, sessions, and processes in SQL Server instance.

Main Capabilities:

- Shows state of active sessions and processes occurring at present.
- Allows tracking server load and determining locks or delays.

Input Parameters:

- [**@loginame =**] { 'login' | session_id | 'ACTIVE' }: User name, session identifier, or special word ACTIVE, showing active processes. Optional parameter.

Output Parameters:

- Table with results, including:
- Session identifier (spid)
- Process status (status)
- User name (loginame)

- Client computer name (hostname)
- Blocking session (blk)
- Active database (dbname)
- Command being executed by process (cmd)

Typical Usage Scenarios:

- Diagnosing performance problems or delays in query processing.
- Monitoring server load and state of current sessions.

Query Examples:

```
-- Get information about all current processes  
EXECUTE sp_who;
```

16.3. Function Descriptions

DATETIME

@@DATEFIRST

Function returns current value of SET DATEFIRST parameter for session.

Main Capabilities:

- Returns number from 1 to 7 corresponding to first day of week
- Value determines which day is considered first day of week
- Affects operation of DATEPART, WEEKDAY and other date functions
- Value can be changed with SET DATEFIRST command

Input Parameters:

- No parameters (system variable)

Output Parameters:

- Integer from 1 to 7:
 1. Monday
 2. Tuesday

3. Wednesday
4. Thursday
5. Friday
6. Saturday
7. Sunday

Typical Usage Scenarios:

- Checking current first day of week settings
- Configuring correct week display in reports
- Ensuring consistent date handling in application
- Debugging queries using week functions

Query Examples:

```
-- 1. Checking current value
SELECT @@DATEFIRST;
-- Result: 7 (if set to default for USA)
```

```
-- 2. Changing and checking value
SET DATEFIRST 1; -- Set Monday as first day
SELECT @@DATEFIRST;
-- Result: 1
```

CURRENT_TIMESTAMP

Function returns current system date and time in datetime format.

Main Capabilities:

- Returns current database server date and time
- Requires no arguments

Input Parameters:

- No parameters

Output Parameters:

- Value of type datetime in format 'YYYY-MM-DD hh:mm:ss.nnn'

Typical Usage Scenarios:

- Recording creation/modification times of records
- Logging events with timestamps
- Comparing with other dates
- Using in filtering conditions

Query Examples:

```
-- 1. Getting current date and time
SELECT CURRENT_TIMESTAMP;
-- Result: '2023-11-15 14:25:36.123' (example)
```

DATE_BUCKET

Function groups date into specified time intervals (buckets).

Main Capabilities:

- Groups dates into specified time intervals
- Supports various time units (day, week, month, etc.)
- Allows specifying interval size (e.g., 2 weeks)
- Can use reference date for start of counting
- Optimized for working with time series

Input Parameters:

- datepart - grouping unit (YEAR, QUARTER, MONTH, WEEK, DAY, etc.)
- number - interval size (positive integer)
- date - date to group
- origin - reference date (optional parameter)

Output Parameters:

- Start date of time interval (bucket)
- Data type corresponds to input date

Typical Usage Scenarios:

- Analyzing time series with grouping
- Building reports with non-standard periods
- Creating time histograms
- Aggregating data into custom periods

Query Examples:

```
-- 1. Grouping by 2 weeks
DECLARE @date DATETIME2 = '2025-04-03 14:21:11';
SELECT DATE_BUCKET(WEEK, 2, @date);
-- Result: '2025-03-31 00:00:00' (start of 2-week interval)
```

DATEADD

Function adds specified time interval to date and returns new date.

Main Capabilities:

- Adds time intervals (years, months, days, hours, etc.) to date
- Supports negative values for subtracting intervals
- Works with various date types (DATE, DATETIME, DATETIME2, SMALLDATETIME)
- Preserves precision of input date

Input Parameters:

- datepart - type of interval (YEAR, QUARTER, MONTH, DAY, WEEK, HOUR, MINUTE, etc.)
- number - number of intervals to add (integer, can be negative)
- date - source date

Output Parameters:

- New date with added interval
- Data type corresponds to input date

Typical Usage Scenarios:

- Calculating expiration dates
- Generating time series
- Determining dates in past or future
- Adjusting timestamps

Query Examples:

```
-- 1. Adding 1 month to date  
SELECT DATEADD(month, 1, '2024-04-08 14:21:31');  
-- Result: '2024-05-08 14:21:31'
```

```
-- 2. Subtracting 10 days  
SELECT DATEADD(day, -10, '2024-04-08');  
-- Result: '2024-03-29'
```

```
-- 3. Adding 3 hours  
SELECT DATEADD(hour, 3, '2024-04-08 14:21:31');  
-- Result: '2024-04-08 17:21:31'
```

DATEDIFF

Function returns difference between two dates in specified time units.

Main Capabilities:

- Calculates difference between two dates
- Supports various units of measurement (years, months, days, minutes, etc.)
- Returns integer (number of complete intervals)
- Ignores smaller time units when calculating

Input Parameters:

- datepart - unit of measurement for difference (YEAR, QUARTER, MONTH, DAY, HOUR, MINUTE, SECOND, etc.)
- startdate - start date
- enddate - end date

Output Parameters:

- Integer difference in specified units
- Always non-negative value (if enddate > startdate)

Typical Usage Scenarios:

- Calculating age or experience
- Determining process duration
- Analyzing time intervals between events
- Checking expiration of deadlines

Query Examples:

```
-- 1. Difference in years (counts only full years)
SELECT DATEDIFF(year, '2005-04-12', '2006-04-12');
-- Result: 1
```

```
-- 2. Difference in months between dates
SELECT DATEDIFF(month, '2023-01-15', '2023-03-15');
-- Result: 2
```

```
-- 3. Exact difference in minutes
SELECT DATEDIFF(minute, '2023-01-01 08:00:00', '2023-01-01 10:30:00');
-- Result: 150
```

DATEDIFF_BIG

Function returns difference between two dates in specified time units, supporting large interval values (bigint).

Main Capabilities:

- Calculates difference between two dates with ability to return very large values
- Supports same units of measurement as DATEDIFF
- Returns value of type bigint (from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- Especially useful for large time intervals or small units of measurement

Input Parameters:

- datepart - unit of measurement for difference (YEAR, QUARTER, MONTH, DAY, HOUR, MINUTE, SECOND, etc.)
- startdate - start date
- enddate - end date

Output Parameters:

- Difference in specified units as bigint
- Can be negative if enddate < startdate

Typical Usage Scenarios:

- Calculating very large time intervals
- Exact calculations in milliseconds or microseconds
- Systems where huge time ranges are possible
- Replaces DATEDIFF when risk of int overflow exists

Query Examples:

```
-- 1. Difference in years (like DATEDIFF, but returns bigint)
SELECT DATEDIFF_BIG(year, '2005-04-12', '2006-04-12');
-- Result: 1
```

```
-- 2. Very large difference in seconds
SELECT DATEDIFF_BIG(second, '1900-01-01', '2100-01-01');
-- Result: 6311433600 (DATEDIFF would cause overflow)
```

```
-- 3. Exact difference in milliseconds
SELECT DATEDIFF_BIG(millisecond, '2023-01-01 00:00:00.000', '2023-01-02
00:00:00.000');
-- Result: 86400000
```

DATEFROMPARTS

Function creates date value from separate numeric components (year, month, day).

Main Capabilities:

- Forms date value from specified parts
- Returns result of type date
- Checks validity of composite date
- Supports standard SQL Server date range (0001-01-01 through 9999-12-31)

Input Parameters:

- year - integer year value (1-9999)
- month - integer month value (1-12)
- day - integer day value (1-31, depending on month)

Output Parameters:

- Date value of type date
- NULL if input parameters form invalid date

Typical Usage Scenarios:

- Assembling date from separate database fields
- Converting numeric values to dates
- Creating dynamic dates in queries
- Validating possible dates

Query Examples:

```
-- 1. Creating standard date
SELECT DATEFROMPARTS(2010, 12, 31);
-- Result: '2010-12-31'
```

```
-- 2. Attempt to create invalid date
SELECT DATEFROMPARTS(2010, 2, 30);
-- Result: Error
```

```
-- 3. Using with variables
DECLARE @y int = 2023, @m int = 5, @d int = 15;
SELECT DATEFROMPARTS(@y, @m, @d);
-- Result: '2023-05-15'
```

DATETIME2FROMPARTS

Function creates datetime2 value from separate numeric components of date and time.

Main Capabilities:

- Forms datetime2 value from specified date and time parts
- Allows setting precision in fractions of seconds
- Checks validity of composite date/time
- Supports full datetime2 range (0001-01-01 through 9999-12-31)

Input Parameters:

- year - year (1-9999)
- month - month (1-12)
- day - day (1-31)
- hour - hours (0-23)
- minute - minutes (0-59)
- second - seconds (0-59)
- fractions - fractions of seconds (0-999999)
- precision - precision (0-5, determines significant digits in fractions)

Output Parameters:

- Value of type datetime2 with specified precision
- NULL if input parameters form invalid date/time

Typical Usage Scenarios:

- Assembling precise timestamps from separate components
- Importing data from systems with separate date/time storage
- Generating test data with controlled precision
- Converting between different date/time formats

Query Examples:

```
-- 1. Creating standard date/time
SELECT DATETIME2FROMPARTS(2010, 12, 31, 23, 59, 59, 0, 0);
-- Result: '2010-12-31 23:59:59'
```

```
-- 2. Creating with millisecond precision
SELECT DATETIME2FROMPARTS(2023, 5, 15, 14, 30, 25, 500, 3);
-- Result: '2023-05-15 14:30:25.500'
```

```
-- 3. Maximum precision (7 digits)
SELECT DATETIME2FROMPARTS(2020, 1, 1, 0, 0, 0, 1234567, 7);
-- Result: '2020-01-01 00:00:00.1234567'
```

```
-- 4. Using with variables
DECLARE @y int = 2023, @ms int = 500;
SELECT DATETIME2FROMPARTS(@y, 1, 1, 0, 0, 0, @ms, 3);
-- Result: '2023-01-01 00:00:00.500'
```

DATETIMEFROMPARTS

Function creates datetime value from separate numeric components of date and time.

Main Capabilities:

- Forms datetime value from specified date and time parts
- Supports range from 1753-01-01 to 9999-12-31
- Checks validity of composite date/time
- Precision to 3.33 milliseconds

Input Parameters:

- year - year (1753-9999)
- month - month (1-12)
- day - day (1-31)
- hour - hours (0-23)
- minute - minutes (0-59)
- second - seconds (0-59)

- millisecond - milliseconds (0-999)

Output Parameters:

- Value of type datetime

Typical Usage Scenarios:

- Assembling date/time from separate fields
- Converting from other date formats
- Generating test data
- Creating timestamps with millisecond precision

Query Examples:

```
-- 1. Creating standard date/time  
SELECT DATETIMEFROMPARTS(2010, 12, 31, 23, 59, 59, 0);  
-- Result: '2010-12-31 23:59:59.000'
```

```
-- 2. Creating with milliseconds  
SELECT DATETIMEFROMPARTS(2023, 5, 15, 14, 30, 25, 500);  
-- Result: '2023-05-15 14:30:25.500'
```

DATETIMEOFFSETFROMPARTS

Function creates datetimeoffset value from separate numeric components of date, time, and time zone offset.

Main Capabilities:

- Forms datetimeoffset value considering time zone
- Supports high precision (up to 100 nanoseconds)
- Checks validity of all date and time components
- Allows setting precision of fractional seconds part

Input Parameters:

- year - year (1-9999)
- month - month (1-12)

- day - day (1-31)
- hour - hours (0-23)
- minute - minutes (0-59)
- second - seconds (0-59)
- fractions - fractions of seconds (0-99999999)
- hour_offset - time zone offset in hours (-14 to +14)
- minute_offset - time zone offset in minutes (0-59)
- precision - precision (0-7)

Output Parameters:

- Value of type datetimeoffset
- NULL if input parameters invalid

Query Examples:

```
-- 1. Creating date with time zone UTC+3
SELECT DATETIMEOFFSETFROMPARTS(2023, 5, 15, 14, 30, 0, 0, 3, 0, 0);
-- Result: '2023-05-15 14:30:00 +03:00'
```

```
-- 2. High precision with UTC-5
SELECT DATETIMEOFFSETFROMPARTS(2023, 1, 1, 0, 0, 0, 1234567, -5, 0, 7);
-- Result: '2023-01-01 00:00:00.1234567 -05:00'
```

```
-- 3. Different time zones
SELECT
    DATETIMEOFFSETFROMPARTS(2023,1,1,0,0,0,0,0,0,0) AS UTC,
    DATETIMEOFFSETFROMPARTS(2023,1,1,0,0,0,0,3,0,0) AS Moscow;
```

DATETRUNC

Function truncates (cuts off) date-time value to specified precision.

Main Capabilities:

- Truncates date-time to given level of precision
- Supports various time units (year, quarter, month, day, etc.)

- Preserves data type of original value
- Works with types date, datetime, datetime2, datetimeoffset

Input Parameters:

- datepart - truncation unit (YEAR, QUARTER, MONTH, DAY, HOUR, MINUTE, SECOND)
- date - source date-time value

Output Parameters:

- Truncated date-time value of same type as input
- Time set to 00:00:00 for corresponding truncation levels

Typical Usage Scenarios:

- Aggregating data by periods
- Comparing dates without considering time
- Building reports with grouping by time periods
- Rounding timestamps

Query Examples:

```
-- 1. Truncation to day (zeroing time)
DECLARE @d datetime2 = '1998-04-12 02:03:04.1234567';
SELECT DATETRUNC(day, @d);
-- Result: '1998-04-12 00:00:00.0000000'
```

```
-- 2. Truncation to month
DECLARE @dt datetime = '2023-07-15 14:30:45';
SELECT DATETRUNC(month, @dt);
-- Result: '2023-07-01 00:00:00.000'
```

```
-- 3. Truncation to hour
DECLARE @dto datetimeoffset = '2025-01-31 23:59:59.9999999 +03:00';
SELECT DATETRUNC(hour, @dto);
-- Result: '2025-02-01 00:00:00.0000000 +03:00'
```

```
-- 4. Truncation to year  
DECLARE @date date = '1999-12-31';  
SELECT DATETRUNC(year, @date);  
-- Result: '1999-01-01'
```

DAY

Function returns day of month from specified date.

Main Capabilities:

- Extracts day number from date value
- Works with all date-time types (date, datetime, datetime2, etc.)
- Always returns integer from 1 to 31

Input Parameters:

- date - date value from which day is extracted

Output Parameters:

- Integer representing day of month (1-31)

Typical Usage Scenarios:

- Extracting day for analysis and reporting
- Filtering by specific days of month
- Using in combination with other date functions
- Grouping data by days of month

Query Examples:

```
-- 1. Extracting day from full date-time  
DECLARE @date1 datetime2 = '2015-04-04 01:01:01.1234567';  
SELECT DAY(@date1);  
-- Result: 4
```

```
-- 2. Extracting day from value of type date
DECLARE @date2 date = '2023-12-31';
SELECT DAY(@date2);
-- Result: 31
```

```
-- 3. Using with date literal
SELECT DAY('1999-01-15');
-- Result: 15
```

```
-- 4. Extracting day from datetimeoffset
DECLARE @date3 datetimeoffset = '2025-07-01 00:00:00 +03:00';
SELECT DAY(@date3);
-- Result: 1
```

EOMONTH

Function returns last day of month for specified date.

Main Capabilities:

- Returns date of last day of month
- Supports optional month offset parameter
- Works with types date, datetime, datetime2, datetimeoffset
- Returns result of same type as input date

Input Parameters:

- start_date - source date
- month_to_add - optional integer number of months to add (can be negative)

Output Parameters:

- Date of last day of month
- NULL if input date is NULL

Typical Usage Scenarios:

- Calculating end of current month
- Determining number of days in month

- Generating monthly reports
- Validating dates

Query Examples:

```
-- 1. Last day of current month  
DECLARE @date DATETIME = '2005-04-12';  
SELECT EOMONTH(@date);  
-- Result: '2005-04-30 00:00:00.000'
```

```
-- 2. Last day of next month  
DECLARE @date2 date = '2023-01-15';  
SELECT EOMONTH(@date2, 1);  
-- Result: '2023-02-28'
```

```
-- 3. Last day of previous month  
DECLARE @date3 datetime2 = '1999-03-10';  
SELECT EOMONTH(@date3, -1);  
-- Result: '1999-02-28 00:00:00.0000000'
```

```
-- 4. For leap year  
DECLARE @date4 date = '2024-02-01';  
SELECT EOMONTH(@date4);  
-- Result: '2024-02-29'
```

GETDATE

Function returns current system date and time.

Main Capabilities:

- Returns current database server date and time
- Precision to 3.33 milliseconds
- Return value type - datetime
- Requires no arguments

Input Parameters:

- No parameters

Output Parameters:

- Current date and time in datetime format ('YYYY-MM-DD hh:mm:ss.nnn')

Typical Usage Scenarios:

- Recording creation/modification times of records
- Logging events
- Comparing with other dates
- Using in filtering conditions

Query Examples:

```
-- 1. Getting current date and time
DECLARE @current_datetime datetime = GETDATE();
SELECT @current_datetime AS CurrentDateTime;
-- Result: '2023-11-20 14:35:22.123'
```

SELECT GETUTCDATE()

GETUTCDATE

Function returns current date and time in UTC format (Coordinated Universal Time).

Main Capabilities:

- Returns current date and time according to Greenwich (UTC)
- Precision to 3.33 milliseconds
- Return value type - datetime
- Requires no arguments
- Independent of server time zone

Input Parameters:

- No parameters

Output Parameters:

- Current date and time UTC in datetime format ('YYYY-MM-DD hh:mm:ss.nnn')

Typical Usage Scenarios:

- Logging events in unified time standard
- Synchronizing data between servers in different time zones
- Comparing timestamps from different sources
- International systems with users in different time zones

Query Examples:

```
-- 1. Getting current date and time UTC
DECLARE @utc_now datetime = GETUTCDATE();
SELECT @utc_now AS UTCDateTime;
-- Result: '2023-11-20 11:40:15.123' (example)
```

```
-- 2. Comparing with local server time
SELECT
    GETDATE() AS LocalTime,
    GETUTCDATE() AS UTCTime,
    DATEDIFF(hour, GETUTCDATE(), GETDATE()) AS TimeZoneDifference;
```

ISDATE

Function checks whether input expression is valid date.

Main Capabilities:

- Checks validity of string representation of date
- Returns 1 (true) for valid dates, 0 (false) - for invalid
- Recognizes various date formats supported by SQL Server

Input Parameters:

- expression - string expression to check

Output Parameters:

- 1 - if expression is valid date
- 0 - if expression is not valid date

Typical Usage Scenarios:

- Validating user input

- Processing data before conversion to date type
- Filtering incorrect records
- Checking date formats in text fields

Query Examples:

```
-- 1. Checking correct date
DECLARE @valid_date varchar(20) = '2023-10-10';
SELECT ISDATE(@valid_date) AS IsValidDate;
-- Result: 1
```

```
-- 2. Checking incorrect date
DECLARE @invalid_date varchar(20) = 'InvalidDate';
SELECT ISDATE(@invalid_date) AS IsValidDate;
-- Result: 0
```

```
-- 3. Checking different date formats
SELECT
    ISDATE('20231010') AS ISOFormat,          -- 1
    ISDATE('10/10/2023') AS USFormat,         -- 1
    ISDATE('10.10.2023') AS EuropeanFormat,   -- 1
    ISDATE('2023-13-01') AS InvalidMonth;     -- 0
```

```
-- 4. Checking NULL value
DECLARE @null_date varchar(20) = NULL;
SELECT ISDATE(@null_date) AS IsValidDate;
-- Result: NULL
```

MONTH Function returns month number from specified date.

Main Capabilities:

- Extracts month number (1-12) from date value
- Works with all date-time types (date, datetime, datetime2, etc.)
- Always returns integer from 1 to 12

Input Parameters:

- date - date value from which month is extracted

Output Parameters:

- Integer representing month of year (1-12)

Typical Usage Scenarios:

- Analyzing data by months
- Grouping and aggregation by months
- Filtering records by specific month
- Using in combination with other date functions

Query Examples:

```
-- 1. Extracting month from full date-time
DECLARE @date1 datetime2 = '2015-04-30 01:01:01.1234567';
SELECT MONTH(@date1);
-- Result: 4
```

```
-- 2. Extracting month from value of type date
DECLARE @date2 date = '2023-12-31';
SELECT MONTH(@date2);
-- Result: 12
```

```
-- 3. Using with date literal
SELECT MONTH('1999-01-15');
-- Result: 1
```

```
-- 4. Extracting month from datetimeoffset
DECLARE @date3 datetimeoffset = '2025-07-01 00:00:00 +03:00';
SELECT MONTH(@date3);
-- Result: 7
```

SYSDATETIME

Function returns current system date and time with high precision.

Main Capabilities:

- Returns current database server date and time

- Returns value of type datetime2
- Requires no arguments

Input Parameters:

- No parameters

Output Parameters:

- Current date and time in datetime2(7) format ('YYYY-MM-DD hh:mm:ss.nnn')

Typical Usage Scenarios:

- Precisely measuring operation execution time
- Recording timestamps with high precision
- Comparing time intervals
- Using in systems with critical time requirements

Query Examples:

```
-- 1. Getting current date and time
DECLARE @current_datetime datetime2(7) = SYSDATETIME();
SELECT @current_datetime AS CurrentDateTime;
-- Result: '2023-11-20 15:42:18.1234567' (example)
```

```
-- 2. Measuring execution time
DECLARE @start_time datetime2(7) = SYSDATETIME();
-- Performing operations
DECLARE @execution_time int = DATEDIFF(millisecond, @start_time, SYSDATETIME());
```

```
-- 3. Comparing with GETDATE()
SELECT
    SYSDATETIME() AS HighPrecisionTime,
    GETDATE() AS StandardTime;
```

SYSDATETIMEOFFSET

Function returns current system date and time considering time zone.

Main Capabilities:

- Returns current date, time and time zone offset of server
- Precision to 100 nanoseconds
- Returns value of type datetimeoffset(6)
- Requires no arguments

Input Parameters:

- No parameters

Output Parameters:

- Current date, time and time zone offset in datetimeoffset format ('YYYY-MM-DD hh:mm:ss.nnnnnnn ±hh:mm')

Typical Usage Scenarios:

- Logging events considering time zone
- Synchronizing data between servers in different locations
- Applications working with international time zones
- Precise time measurements with time zone binding

Query Examples:

```
-- 1. Getting current date, time and time zone
DECLARE @current_datetimeoffset datetimeoffset(7) = SYSDATETIMEOFFSET();
SELECT @current_datetimeoffset AS CurrentDateTimeOffset;
-- Result: '2023-11-20 15:45:22.1234567 +03:00' (example)
```

```
-- 2. Comparing with UTC time
SELECT
    SYSDATETIMEOFFSET() AS LocalTimeWithOffset,
    SYSUTCDATETIME() AS UTCTime;
```

SYSUTCDATETIME

Function returns current system date and time in UTC format (Coordinated Universal Time) with high precision.

Main Capabilities:

- Returns current date and time according to Greenwich (UTC)
- Precision to 100 nanoseconds
- Returns value of type datetime2(7)
- Independent of server time zone
- Requires no arguments

Input Parameters:

- No parameters

Output Parameters:

- Current date and time UTC in datetime2(7) format ('YYYY-MM-DD hh:mm:ss.nnnnnnn')

Typical Usage Scenarios:

- International systems with unified time standard
- Precise time measurements in UTC
- Synchronizing data between servers
- Audit records with UTC timestamps

Query Examples:

```
-- 1. Getting current date and time UTC
DECLARE @utc_now datetime2(7) = SYSUTCDATETIME();
SELECT @utc_now AS UTCDateTime;
-- Result: '2023-11-20 12:48:15.1234567' (example)
```

```
-- 2. Comparing with local server time
SELECT
    SYSUTCDATETIME() AS UTCTime,
    SYSDATETIME() AS LocalTime,
    DATEDIFF(minute, SYSUTCDATETIME(), SYSDATETIME()) AS TimeZoneDiff;
```

TIMEFROMPARTS

Function creates time value from separate numeric components.

Main Capabilities:

- Forms time value from specified parts
- Allows setting precision in fractions of seconds
- Checks validity of composite time

Input Parameters:

- hour - hours (0-23)
- minute - minutes (0-59)
- second - seconds (0-59)
- fractions - fractions of seconds (0-99999999)
- precision - precision (0-7, determines significant digits in fractions)

Output Parameters:

- Value of type time with specified precision
- NULL if input parameters invalid

Typical Usage Scenarios:

- Assembling time from separate database fields
- Converting numeric values to time format
- Generating test data with controlled precision
- Validating possible time values

Query Examples:

```
-- 1. Creating standard time
DECLARE @time1 time = TIMEFROMPARTS(23, 59, 59, 0, 0);
SELECT @time1 AS FullTime;
-- Result: '23:59:59'
```

TODATETIMEOFFSET

Function converts datetime2 value to datetimeoffset, adding specified time zone offset.

Main Capabilities:

- Converts datetime2 to datetimeoffset
- Adds specified time zone offset
- Preserves original date-time precision
- Does not change time value itself, only adds offset

Input Parameters:

- date - source date-time value (datetime2)
- offset - time zone offset in minutes (integer from -840 to +840, corresponding to -14:00 to +14:00)

Output Parameters:

- Value of type datetimeoffset
- NULL if input parameters invalid

Typical Usage Scenarios:

- Adding time zone information to existing timestamps
- Converting local time to format with offset specification
- Preparing data for systems considering time zones
- Synchronizing data between servers in different time zones

Query Examples:

```
-- 1. Adding positive offset (UTC+3)
DECLARE @local_time datetime2 = SYSDATETIME();
SELECT TODATETIMEOFFSET(@local_time, 180) AS MoscowTime;
-- Result: '2023-11-21 14:30:45.123 +03:00'
```

```
-- 2. Adding negative offset (UTC-5)
DECLARE @server_time datetime2 = '2023-12-31 23:59:59';
SELECT TODATETIMEOFFSET(@server_time, -300) AS NewYorkTime;
-- Result: '2023-12-31 23:59:59.000 -05:00'
```

```
-- 3. Converting with minute precision
DECLARE @precise_time datetime2(3) = '2024-01-01 00:00:00.123';
SELECT TODATETIMEOFFSET(@precise_time, 0) AS UTCtime;
-- Result: '2024-01-01 00:00:00.123 +00:00'
```

YEAR

Function returns year from specified date.

Main Capabilities:

- Extracts year from date value
- Works with all date-time types (date, datetime, datetime2, etc.)
- Returns four-digit year (0001-9999)
- Independent of server localization settings

Input Parameters:

- date - date value from which year is extracted

Output Parameters:

- Integer representing year (0001-9999)

Typical Usage Scenarios:

- Analyzing data by years
- Filtering records by year
- Grouping data by years
- Using in combination with other date functions

Query Examples:

```
-- 1. Extracting year from full date-time
DECLARE @date1 datetime2 = '2010-04-04T01:01:01.1234567';
SELECT YEAR(@date1);
-- Result: 2010
```

```
-- 2. Extracting year from value of type date  
DECLARE @date2 date = '1999-12-31';  
SELECT YEAR(@date2);  
-- Result: 1999
```

```
-- 3. Using with date literal  
SELECT YEAR('2025-01-01');  
-- Result: 2025
```

```
-- 4. Extracting year from datetimeoffset  
DECLARE @date3 datetimeoffset = '0001-01-01 00:00:00 +00:00';  
SELECT YEAR(@date3);  
-- Result: 1
```

LOGICAL

CHOOSE

Function returns element from list of values by specified index.

Main Capabilities:

- Returns value from list by specified position
- Supports various data types in list
- Returns NULL if index out of range
- Element numbering starts from 1

Input Parameters:

- index - position of element to return (INTEGER)
- val1, val2,... - list of values (any compatible data types)

Output Parameters:

- Value of same type as list elements
- NULL if index less than 1 or greater than number of elements

Typical Usage Scenarios:

- Getting element from fixed list by position
- Replacing complex CASE WHEN expressions
- Dynamic value selection
- Forming results based on numeric indexes

Query Examples:

```
-- 1. Selecting string value  
SELECT CHOOSE(3, 'January', 'February', 'March', 'April');  
-- Result: 'March'
```

```
-- 2. Selecting numeric value  
SELECT CHOOSE(1, 100, 200, 300);  
-- Result: 100
```

```
-- 3. Index out of range  
SELECT CHOOSE(0, 'A', 'B', 'C');  
-- Result: NULL
```

```
SELECT CHOOSE(5, 'A', 'B', 'C');  
-- Result: NULL
```

GREATEST

Function returns greatest value from list of passed arguments.

Main Capabilities:

- Returns maximum value from list of arguments
- Supports comparison of various data types (but arguments must be comparable)
- Returns NULL if any argument is NULL
- For strings uses current database collation rules
- Works with numbers, dates, strings, and other comparable types

Input Parameters:

- arg1, arg2, ... - list of values to compare (2 or more arguments)

- Supported types: numeric (INT, DECIMAL, FLOAT, etc.), string (CHAR, VARCHAR), dates (DATE, DATETIME), BIT

Output Parameters:

- Greatest value from passed (type corresponds to argument types)
- NULL if any argument is NULL

Typical Usage Scenarios:

- Comparing values from different columns of same row
- Determining extreme dates in set
- Selecting maximum value from alternative options
- Simplifying complex conditions in queries
- Processing data with dynamically defined boundaries

Query Examples:

```
-- 1. Comparing different numeric types
SELECT GREATEST(3.14, 10, 15.67, -1);
-- Result: 15.67
```

```
-- 2. Selecting latest date
SELECT GREATEST(
    '2023-01-15',
    '2023-03-10',
    '2022-12-01',
    GETDATE() -- current date
);
-- Result: Current date
```

```
-- 3. Comparing strings case-sensitive
SELECT GREATEST('Apple', 'apple', 'APPLE');
-- Result: Apple
```

IIF

Function returns one of two values depending on logical condition evaluation.

Main Capabilities:

- Returns value when condition true or false
- Supports various data types for return values
- Simplifies queries compared to CASE WHEN
- Can be used anywhere expression is allowed

Input Parameters:

- condition - logical expression to check (BOOLEAN)
- true_value - value returned when condition true (any type)
- false_value - value returned when condition false (compatible type)

Output Parameters:

- Value true_value or false_value depending on condition
- NULL if condition evaluates to NULL

Typical Usage Scenarios:

- Simple conditional value returns
- Replacing simple CASE WHEN expressions
- Conditional result formatting
- Logical checks in computed fields

Query Examples:

```
-- 1. Comparing numbers
SELECT IIF(40000 > 50000, 'TRUE', 'FALSE');
-- Result: 'FALSE'
```

```
-- 2. Checking equality
SELECT IIF(40000 = 50000, 'TRUE', 'FALSE');
-- Result: 'FALSE'
```

LEAST

Function returns smallest value from list of passed arguments.

Main Capabilities:

- Returns minimum value from list of arguments
- Supports comparison of various data types (arguments must be compatible)
- Returns NULL if any argument is NULL
- For strings uses current database collation rules

Input Parameters:

- arg1, arg2, ... - list of values to compare (minimum 2 arguments)
- Supported types: numeric, string, dates

Output Parameters:

- Smallest value from passed (type corresponds to argument types)
- NULL if any argument is NULL

Typical Usage Scenarios:

- Finding minimum value among several columns
- Comparing alternative values
- Simplifying complex CASE WHEN expressions
- Determining earliest dates in set

Query Examples:

```
-- 1. Selecting smallest numeric value
SELECT LEAST(1, 32322, 2);
-- Result: 1
```

```
-- 2. Selecting smallest string (alphabetical comparison)
SELECT LEAST('a', 'AAAAA', '', 'aaa');
-- Result: '' (empty string)
```

```
-- 3. Comparing dates
SELECT LEAST('2023-01-15', '2023-03-10', '2022-12-01');
-- Result: '2022-12-01'
```

MATHEMATICAL

ABS

Function returns absolute value (modulus) of specified numeric expression.

Main Capabilities:

- Returns non-negative value of number
- Works with various numeric data types
- Preserves type of original value
- Supports integers and fractions

Input Parameters:

- numeric_expression - numeric expression (int, decimal, float, money, etc.)

Output Parameters:

- Absolute value of same type as input
- NULL if input value NULL

Typical Usage Scenarios:

- Calculating distances or differences without considering direction
- Processing financial indicators
- Normalizing numeric values
- Mathematical calculations requiring non-negative values

Query Examples:

```
-- 1. Integer
SELECT ABS(-5) AS AbsoluteValue;
-- Result: 5
```

```
-- 2. Fractional number
SELECT ABS(-12.34) AS AbsoluteValue;
-- Result: 12.34
```

```
-- 3. Zero value
SELECT ABS(0) AS AbsoluteValue;
-- Result: 0
```

```
-- 4. Already positive number
SELECT ABS(100) AS AbsoluteValue;
-- Result: 100
```

```
-- 5. Different data types
SELECT
    ABS(CAST(-10 AS int)) AS IntValue,
    ABS(CAST(-3.14 AS decimal(5,2))) AS DecimalValue,
    ABS(CAST(-1.5E+4 AS float)) AS FloatValue;
```

ACOS

Function returns arccosine of specified numeric expression in radians.

Main Capabilities:

- Calculates angle in radians whose cosine equals given value
- Input value range: from -1 to 1
- Returns value in range from 0 to π (pi) radians
- Supports float and decimal types

Input Parameters:

- numeric_expression - numeric expression ($-1 \leq x \leq 1$)

Output Parameters:

- Value in radians (float type)

Typical Usage Scenarios:

- Trigonometric calculations
- Processing geometric data
- Scientific and engineering calculations
- Graphical transformations

Query Examples:

```
-- 1. Basic calculation
SELECT ACOS(0.5) AS ArcCosine;
-- Result: 1.0471975511966 ( $\pi/3 \approx 60^\circ$ )
```

```
-- 2. Converting to degrees
SELECT ACOS(0.5) * 180 / PI() AS Degrees;
-- Result: 60
```

ASIN

Function returns arcsine of specified numeric expression in radians.

Main Capabilities:

- Calculates angle in radians whose sine equals given value
- Input value range: from -1 to 1
- Returns value in range from $-\pi/2$ to $\pi/2$ radians
- Supports float and decimal types

Input Parameters:

- numeric_expression - numeric expression ($-1 \leq x \leq 1$)

Output Parameters:

- Value in radians (float type)
- NULL if input value NULL or out of range

Typical Usage Scenarios:

- Trigonometric calculations in mathematics and physics
- Processing geometric data
- Engineering calculations
- Scientific research

Query Examples:

```
-- 1. Basic calculation
SELECT ASIN(0.5) AS ArcSine;
-- Result: 0.5235987755982989 ( $\pi/6 \approx 30^\circ$ )
```

```
-- 2. Converting to degrees
SELECT ASIN(0.5) * 180 / PI() AS Degrees;
-- Result: 30
```

ATAN

Function returns arctangent (inverse tangent) of specified numeric expression in radians.

Main Capabilities:

- Calculates angle in radians whose tangent equals given value
- Accepts any numeric values (from $-\infty$ to $+\infty$)
- Returns value in range from $-\pi/2$ to $\pi/2$ radians
- Supports float and decimal types

Input Parameters:

- numeric_expression - numeric expression (any real number)

Output Parameters:

- Value in radians (float type)
- NULL if input value NULL

Typical Usage Scenarios:

- Trigonometric calculations
- Processing geometric data
- Calculating slope angles
- Scientific and engineering calculations

Query Examples:

```
-- 1. Basic calculation
SELECT ATAN(0.5) AS ArcTangent;
-- Result: 0.4636476090008061 (≈26.565°)
```

```
-- 2. Converting to degrees
SELECT ATAN(1) * 180 / PI() AS Degrees;
-- Result: 45
```

```
-- 3. Boundary values
SELECT
    ATAN(0) AS ZeroRadians, -- 0
    ATAN(1E10) AS ApproxPiHalves; -- ≈1.5707963267949 (π/2)
```

ATN2

Function returns arctangent (angle) between positive X-axis and ray from origin to point (y, x) in radians.

Main Capabilities:

- Calculates angle in radians for given coordinates (y, x)
- Considers signs of both parameters for determining correct quadrant
- Returns value in range from $-\pi$ to π radians
- Supports float and decimal types

Input Parameters:

- y - y coordinate (ordinate)
- x - x coordinate (abscissa)

Output Parameters:

- Angle in radians (float type)
- NULL if both input values NULL

Typical Usage Scenarios:

- Determining angle by coordinates
- Converting Cartesian coordinates to polar

- Calculating directions in geometric problems
- Graphical and game calculations

Query Examples:

```
-- 1. Basic calculation (45 degrees)
SELECT ATN2(1, 1) AS ArcTangent;
-- Result: 0.7853981633974483 ( $\pi/4 \approx 45^\circ$ )
```

```
-- 2. Different quadrants
SELECT
    ATN2(1, 1) AS Q1, -- first quadrant
    ATN2(1, -1) AS Q2, -- second quadrant
    ATN2(-1, -1) AS Q3, -- third quadrant
    ATN2(-1, 1) AS Q4; -- fourth quadrant
```

```
-- 3. Converting to degrees
SELECT ATN2(1, 0) * 180 / PI() AS Degrees;
-- Result: 90
```

```
-- 4. Special cases
SELECT
    ATN2(0, 1) AS ZeroY, -- 0
    ATN2(1, 0) AS PiHalves; --  $\pi/2 \approx 1.5707963267949$ 
```

CEILING

Function returns smallest integer that is greater than or equal to specified numeric expression.

Main Capabilities:

- Rounds number up to nearest integer
- Works with various numeric types (int, decimal, float, money)
- Preserves data type of input value
- For negative numbers rounds toward zero

Input Parameters:

- `numeric_expression` - numeric expression to round

Output Parameters:

- Rounded value of same type as input
- NULL if input value NULL

Typical Usage Scenarios:

- Financial calculations (rounding in favor of seller)
- Calculating number of pages/blocks
- Rounding division results
- Systems requiring integer values

Query Examples:

```
-- 1. Rounding positive number
SELECT CEILING(45) AS RoundedValue;
-- Result: 45
```

```
-- 2. Rounding fractional number
SELECT CEILING(45.12) AS RoundedValue;
-- Result: 46
```

```
-- 3. Rounding negative number
SELECT CEILING(-45.12) AS RoundedValue;
-- Result: -45
```

```
-- 4. Different data types
SELECT
    CEILING(123.456) AS DecimalValue,
    CEILING(78.9) AS FloatValue,
    CEILING($19.99) AS MoneyValue;
```

```
-- 5. Practical application
DECLARE @items int = 100, @per_page int = 9;
SELECT CEILING(@items * 1.0 / @per_page) AS PagesNeeded;
-- Result: 12
```

COS

Function returns trigonometric cosine of specified angle in radians.

Main Capabilities:

- Calculates cosine of angle
- Accepts angle in radians
- Returns value from -1 to 1
- Supports float and decimal types

Input Parameters:

- angle - angle in radians (any real number)

Output Parameters:

- Cosine of angle (float type)
- NULL if input value NULL

Query Examples:

```
without tables:
-- 1. Basic calculation (cosine of  $\pi/4$  radians = 45 degrees)
SELECT COS(PI()/4) AS CosineValue;
-- Result: 0.7071067811865476 (2/2)
```

```
-- 2. Cosine of 60 degrees (with degree to radian conversion)
SELECT COS(60 * PI()/180) AS Cosine60Deg;
-- Result: 0.5
```

```
-- 3. Cosine of 0 and  $\pi$  radians
SELECT
    COS(0) AS Cos0,          -- 1.0
    COS(PI()) AS CosPi;     -- -1.0
```

```
-- 4. Using in mathematical expression
DECLARE @angle FLOAT = PI()/3; -- 60 degrees in radians
SELECT
    @angle AS AngleRadians,
    COS(@angle) AS Cosine,
    SIN(@angle) AS Sine,
    (COS(@angle) * COS(@angle)) + (SIN(@angle) * SIN(@angle)) AS TrigIdentity;
-- Last column should be 1.0 (verification of trigonometric identity)
```

```
-- 5. Boundary values
SELECT
    COS(0) AS ZeroRadians,      -- 1.0
    COS(PI()/2) AS PiHalvesRad, -- 0.0
    COS(PI()) AS PiRadians;    -- -1.0
```

COT

Function returns trigonometric cotangent of specified angle in radians.

Main Capabilities:

- Calculates cotangent of angle ($1/\tan(\text{angle})$)
- Accepts angle in radians
- Supports float and decimal types
- Returns NULL for angles multiples of π (where $\tan(\text{angle}) = 0$)

Input Parameters:

- angle - angle in radians (any real number, except πn , where n is integer)

Output Parameters:

- Cotangent of angle (float type)
- NULL if input value NULL or angle multiple of π

Query Examples:

```
-- 1. Basic calculation (cotangent of  $\pi/4$  radians = 45 degrees)
SELECT COT(PI()/4) AS CotangentValue;
-- Result: 1.0 (since  $\tan(\pi/4) = 1$ , and  $\cot = 1/\tan$ )
```

```
-- 2. Cotangent of 30 degrees ( $\pi/6$  radians)
SELECT COT(PI()/6) AS Cot30Deg;
-- Result: 1.7320508075688772 (3)
```

```
-- 3. Using variable
DECLARE @angle FLOAT = PI()/3;
```

—4. Usage in a mathematical expression DECLARE @angle FLOAT = PI()/3; — 60 degrees in radians SELECT @angle AS AngleRadians, COS(@angle) AS Cosine, SIN(@angle) AS Sine, (COS(@angle) * COS(@angle)) + (SIN(@angle) * SIN(@angle)) AS TrigIdentity; — The last column should be ≈ 1.0 (check of trigonometric identity)

```
-- 5. Boundary values
SELECT
    COS(0) AS ZeroRadians,      -- 1.0
    COS(PI()/2) AS PiHalvesRad, -- 0.0
    COS(PI()) AS PiRadians;    -- -1.0
```

COT

The function returns the trigonometric cotangent of the specified angle in radians.

Main capabilities:

- Calculates the cotangent of an angle ($1/\tan(\text{angle})$)
- Accepts an angle in radians
- Supports float and decimal types
- Returns NULL for angles that are multiples of π (where $\tan(\text{angle}) = 0$)

Input parameters:

- angle - angle in radians (any real number except πn , where n is an integer)

Output parameters:

- Cotangent of the angle (float type)
- NULL if the input value is NULL or the angle is a multiple of π

Query examples:

```
-- 1. Basic calculation (cotangent of  $\pi/4$  radians = 45 degrees)
SELECT COT(PI()/4) AS CotangentValue;
-- Result: 1.0 (since  $\tan(\pi/4) = 1$ , and  $\cot = 1/\tan$ )
```

```
-- 2. Cotangent of 30 degrees ( $\pi/6$  radians)
SELECT COT(PI()/6) AS Cot30Deg;
-- Result: 1.7320508075688772 (3)
```

```
-- 3. Using a variable
DECLARE @angle FLOAT = PI()/3; -- 60 degrees
SELECT
    @angle AS AngleRadians,
    COT(@angle) AS Cotangent,
    1.0/TAN(@angle) AS ManualCot; -- Check via tangent
```

```
-- 4. Conversion from degrees
SELECT COT(45 * PI()/180) AS Cot45Deg;
-- Result: 1.0
```

COUNT

The function returns the number of items in a group.

Main capabilities:

- Counts the number of rows or non-NULL values
- Works with all data types
- Supports DISTINCT for counting unique values
- Has three usage variants:
- COUNT(*) - counts all rows
- COUNT(column) - counts non-NULL values in a column
- COUNT(DISTINCT column) - counts unique non-NULL values

Input parameters:

- expression - column or expression to count (optional)
- * - special parameter for counting all rows

Output parameters:

- Integer - number of items
- Always returns a value (even 0), except for cases with NULL in aggregate

Typical usage scenarios:

- Counting the total number of records
- Checking for data presence
- Statistical aggregation
- Counting unique values

Query examples:

```
-- 1. Create a temporary employee table
CREATE TABLE #Employees (
    ID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Department NVARCHAR(50)
);
INSERT INTO #Employees VALUES (1, 'Ivan Ivanov', 'IT'), (2, 'Petr Petrov', 'HR'), (3,
'Sergey Sergeev', 'IT');
SELECT COUNT(*) AS TotalEmployees FROM #Employees;
-- Result: 3
DROP TABLE #Employees;
```

```
-- 2. Create a temporary product table
CREATE TABLE #Products (
    ID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Price DECIMAL(10,2),
    Category NVARCHAR(50)
);
INSERT INTO #Products VALUES
(1, 'Mouse', 1200.00, 'Components'),
(2, NULL, 35000.00, 'Laptops'),
(3, 'Keyboard', NULL, 'Components');
SELECT COUNT(Name) AS NamedProducts FROM #Products;
-- Result: 2
DROP TABLE #Products;
```

```
-- 3. Create a temporary orders table
CREATE TABLE #Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    Amount DECIMAL(10,2)
);
INSERT INTO #Orders VALUES
(1, 101, '2023-01-15', 1500.00),
(2, 102, '2023-01-15', 2300.00),
(3, 101, '2023-01-16', 800.00),
(4, 103, '2023-01-16', 4500.00),
(5, 102, '2023-01-17', 1200.00);
SELECT COUNT(DISTINCT CustomerID) AS UniqueCustomers FROM #Orders;
-- Result: 3
DROP TABLE #Orders;
```

```
-- 4. Complex example with grouping
CREATE TABLE #Sales (
    ID INT PRIMARY KEY,
    Product NVARCHAR(100),
    Region NVARCHAR(50),
    Amount DECIMAL(10,2)
);
INSERT INTO #Sales VALUES
(1, 'TV', 'Moscow', 45000.00),
(2, 'Phone', 'Moscow', 35000.00),
(3, 'TV', 'SPb', 42000.00),
(4, 'Laptop', 'Kazan', 65000.00),
(5, 'Phone', 'Moscow', 32000.00);
SELECT
    Region,
    COUNT(*) AS TotalSales,
    COUNT(DISTINCT Product) AS UniqueProducts
FROM #Sales
GROUP BY Region;
DROP TABLE #Sales;
```

COUNT_BIG

The function returns the number of items in a group as a BIGINT value.

Main capabilities:

- Similar to COUNT, but returns BIGINT instead of INT
- Suitable for very large tables (more than 2 billion records)
- Supports the same usage variants as COUNT:
- COUNT_BIG(*)
- COUNT_BIG(column)
- COUNT_BIG(DISTINCT column)

Input parameters:

- expression - column or expression to count (optional)
- * - special parameter for counting all rows

Output parameters:

- Number of items as BIGINT
- Value range: from 0 to 9,223,372,036,854,775,807

Typical usage scenarios:

- Counting rows in very large tables
- Data aggregation in large-volume warehouses
- When guaranteed avoidance of INT overflow is required

Query examples:

```
-- 1. Create a table with a potentially large number of records
CREATE TABLE BigData (
    ID BIGINT IDENTITY(1,1),
    DataValue NVARCHAR(100),
    CreatedDate DATETIME DEFAULT GETDATE()
);
-- Insert test data (in reality there could be billions of records)
INSERT INTO BigData (DataValue)
SELECT 'Value ' + CAST(ROW_NUMBER() OVER(ORDER BY (SELECT NULL)) AS NVARCHAR(10))
FROM sys.objects a, sys.objects b, sys.objects c;
-- Counting using COUNT_BIG
SELECT COUNT_BIG(*) AS TotalRows FROM BigData;
```

```
-- 2. Create a transactions table
CREATE TABLE Transactions (
    TransactionID BIGINT PRIMARY KEY,
    AccountID INT,
    Amount DECIMAL(19,4),
    TransactionDate DATETIME2
);
-- Insert test data
INSERT INTO Transactions VALUES
(1, 1001, 1500.00, '2023-01-01'),
(2, 1002, 2300.50, '2023-01-02'),
(3, 1001, 800.25, '2023-01-02'),
(4, 1003, 4500.75, '2023-01-03');
-- Count unique accounts with transactions
SELECT COUNT_BIG(DISTINCT AccountID) AS UniqueAccounts FROM Transactions;
```

```
-- 3. Example with GROUP BY
SELECT
    YEAR(TransactionDate) AS Year,
    COUNT_BIG(*) AS TransactionCount
FROM Transactions
GROUP BY YEAR(TransactionDate);
```

DEGREES

The function converts a value from radians to degrees.

Main capabilities:

- Converts radians to degrees
- Accepts a numeric expression (radians)
- Returns a value of type float

Input parameters:

- `numeric_expression` - value in radians (type float, decimal, or compatible numeric type)

Output parameters:

- Value in degrees (type float)
- NULL if the input value is NULL

Typical usage scenarios:

- Converting radians to degrees in trigonometric calculations
- Converting results of mathematical functions (ASIN, ACOS, ATAN) to degrees
- Working with geometric data where degrees are required
- Visualizing angles in degrees in reports and interfaces
- Setting parameters in graphical systems using degree measure

Query examples:

```
-- 1. Basic conversion of  $\pi$  radians to degrees
SELECT DEGREES(PI()) AS Degrees;
-- Result: 180.0
```

```
-- 2. Conversion of  $\pi/2$  radians
SELECT DEGREES(PI()/2) AS Degrees90;
-- Result: 90.0
```

```
-- 3. Conversion of 1 radian
SELECT DEGREES(1) AS Degrees1Rad;
-- Result: 57.29577951308232
```

```
-- 4. Special values
SELECT
    DEGREES(0) AS ZeroRadians,    -- 0.0
    DEGREES(PI()/4) AS Degrees45; -- 45.0
```

```
-- 5. Using a variable
DECLARE @radians FLOAT = 1.5;
SELECT DEGREES(@radians) AS DegreesResult;
```

EXP

The function returns the exponent of the specified numeric expression (e to the power of x).

Main capabilities:

- Calculates the value of the exponent (e^x)
- Supports float and decimal types
- Inverse function - LOG() (natural logarithm)

Input parameters:

- numeric_expression - exponent (any real number)

Output parameters:

- Exponent value (type float)
- NULL if the input value is NULL

Typical usage scenarios:

- Financial calculations of compound interest and investment growth
- Modeling exponential growth in natural sciences
- Statistical calculations (normal distribution, logistic models)
- Machine learning (neural network activation functions)
- Physical calculations (radioactive decay, thermal processes)

Query examples:

```
-- 1. Basic calculation (e^1)
SELECT EXP(1) AS Exponential;
-- Result: 2.718281828459045
```

```
-- 2. Calculation of e^0
SELECT EXP(0) AS Exponential;
-- Result: 1.0
```

```
-- 3. Calculation of e^2.5
SELECT EXP(2.5) AS Exponential;
-- Result: 12.18249396070347
```

```
-- 4. Using a variable  
DECLARE @x FLOAT = 1.5;  
SELECT EXP(@x) AS Result;  
-- Result: 4.4816890703380645
```

```
-- 5. Special values  
SELECT  
    EXP(1) AS E,  
    EXP(-1) AS InverseE; -- 0.36787944117144233
```

FLOOR

The function returns the largest integer less than or equal to the specified numeric expression.

Main capabilities:

- Rounds a number down to the nearest integer
- Works with various numeric types (int, decimal, float, money)
- Preserves the data type of the input value
- For negative numbers, rounds towards negative infinity

Input parameters:

- `numeric_expression` - numeric expression to round

Output parameters:

- Rounded value of the same type as the input
- NULL if the input value is NULL

Typical usage scenarios:

- Financial calculations with rounding down (taxes, payments)
- Packaging and logistics (calculating the minimum amount of packaging)
- Building histograms and data discretization
- Processing time intervals (full days, weeks, months)
- Game logic (splitting resources into integer parts)

Query examples:

```
-- 1. Rounding a positive number
SELECT FLOOR(123.49) AS RoundedValue;
-- Result: 123
```

```
-- 2. Rounding a negative number
SELECT FLOOR(-123.49) AS RoundedValue;
-- Result: -124
```

```
-- 3. Different data types
SELECT
    FLOOR(123.99) AS DecimalValue,
    FLOOR(78.9) AS FloatValue,
    FLOOR($19.99) AS MoneyValue;
```

```
-- 4. Practical application (calculation of full boxes)
DECLARE @totalItems INT = 100, @itemsPerBox INT = 9;
SELECT FLOOR(@totalItems * 1.0 / @itemsPerBox) AS FullBoxes;
-- Result: 11
```

```
-- 5. Comparison with CEILING
SELECT
    FLOOR(45.12) AS FloorValue, -- 45
    CEILING(45.12) AS CeilValue; -- 46
```

LOG

The function returns the natural logarithm (base e) of the specified numeric expression.

Main capabilities:

- Calculates the natural logarithm (ln(x))
- Supports float and decimal types
- Inverse function - EXP() (exponent)
- Can specify an arbitrary logarithm base

Input parameters:

- base - optional logarithm base (default is $e \approx 2.71828$)
- numeric_expression - positive number (greater than 0)

Output parameters:

- Logarithm value (type float)
- NULL if the input value is NULL or ≤ 0

Typical usage scenarios:

- Calculating investment/deposit growth periods
- Data analysis (comparing large numbers)
- Setting up logarithmic scales on graphs
- Checking mathematical calculations

Query examples:

```
-- 1. Natural logarithm (base e)
SELECT LOG(10) AS NaturalLog;
-- Result: 2.302585092994046
```

```
-- 2. Logarithm base 10
SELECT LOG(10, 10) AS Log10;
-- Result: 1.0
```

```
-- 3. Logarithm with arbitrary base (2)
SELECT LOG(2, 8) AS Log2_8;
-- Result: 3.0
```

```
-- 4. Special values
SELECT
    LOG(EXP(1)) AS LnE,  -- 1.0
    LOG(1) AS Ln1;      -- 0.0
```

LOG10

The function returns the decimal logarithm (base 10) of the specified numeric expression.

Main capabilities:

- Calculates the logarithm base 10 (log₁₀)
- Supports float and decimal types
- Inverse function - POWER(10, x)
- Specialized version of the LOG function for base 10

Input parameters:

- numeric_expression - positive number (greater than 0)

Output parameters:

- Logarithm value (type float)
- NULL if the input value is NULL or ≤ 0

Typical usage scenarios:

- Determining the number of digits in numbers (digit count)
- Comparing very large numbers through their orders of magnitude
- Analyzing data with exponential growth (sales, traffic)

Query examples:

```
-- 1. Decimal logarithm of 100
SELECT LOG10(100) AS Log100;
-- Result: 2.0
```

```
-- 2. Decimal logarithm of 1
SELECT LOG10(1) AS Log1;
-- Result: 0.0
```

```
-- 3. Decimal logarithm of 0.1
SELECT LOG10(0.1) AS Log01;
-- Result: -1.0
```



```
-- 4. Using a variable  
DECLARE @value FLOAT = 1000;  
SELECT LOG10(@value) AS LogResult;  
-- Result: 3.0
```

MAX

The function returns the maximum value in a set of values.

Main capabilities:

- Finds the maximum value in a column or expression
- Works with numeric types, dates, strings
- Can be used with GROUP BY for group analysis

Input parameters:

- expression - column or expression to analyze
- ALL | DISTINCT - optional modifier (default is ALL)

Output parameters:

- Maximum value of the same type as the input
- NULL if all values are NULL

Typical usage scenarios:

- Determining the maximum price of products in a category
- Finding the latest order or update date
- Identifying record values (max sales, max traffic)
- Analyzing peak loads (max connections, max traffic)
- Finding the maximum age/experience in a sample

Query examples:

```
-- 1. Create a Products table and find the maximum price
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName NVARCHAR(100),
    Price DECIMAL(10,2),
    Category NVARCHAR(50)
);
INSERT INTO Products VALUES
(1, 'Milk', 85.50, 'Groceries'),
(2, 'Bread', 45.00, 'Groceries'),
(3, 'TV', 45000.00, 'Electronics'),
(4, 'Smartphone', 78000.00, 'Electronics');
SELECT MAX(Price) AS MaxPrice FROM Products;
-- Result: 78000.00
```

```
-- 2. Maximum date in the Orders table
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    Amount DECIMAL(10,2)
);
INSERT INTO Orders VALUES
(1, '2023-01-15', 1500.00),
(2, '2023-02-20', 2300.00),
(3, '2023-01-10', 800.00);
SELECT MAX(OrderDate) AS LatestOrder FROM Orders;
-- Result: '2023-02-20'
```

```
-- 3. Grouping by categories
SELECT
    Category,
    MAX(Price) AS MaxCategoryPrice
FROM Products
GROUP BY Category;
```

```
-- 4. Maximum string (lexicographic comparison)
SELECT MAX(ProductName) AS LastProductName FROM Products;
-- Result: 'Bread' (last alphabetically)
```

MIN

The function returns the minimum value in a set of values.

Main capabilities:

- Finds the minimum value in a column or expression
- Works with numeric types, dates, strings
- Ignores NULL values
- Can be used with GROUP BY for group analysis

Input parameters:

- expression - column or expression to analyze
- ALL | DISTINCT - optional modifier (default is ALL)

Output parameters:

- Minimum value of the same type as the input
- NULL if all values are NULL

Typical usage scenarios:

- Finding the minimum cost of products
- Determining the earliest date of an event
- Identifying minimum values (min temperature, min balance)
- Analyzing minimum task completion times
- Finding the lowest rating or score

Query examples:

```
-- 1. Create a Products table and find the minimum price
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName NVARCHAR(100),
    Price DECIMAL(10,2),
    Category NVARCHAR(50)
);
INSERT INTO Products VALUES
(1, 'Milk', 85.50, 'Groceries'),
(2, 'Bread', 45.00, 'Groceries'),
(3, 'TV', 45000.00, 'Electronics'),
(4, 'Smartphone', 78000.00, 'Electronics');
SELECT MIN(Price) AS MinPrice FROM Products;
-- Result: 45.00
```

```
-- 2. Minimum date in the Orders table
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    Amount DECIMAL(10,2)
);
INSERT INTO Orders VALUES
(1, '2023-01-15', 1500.00),
(2, '2023-02-20', 2300.00),
(3, '2023-01-10', 800.00);
SELECT MIN(OrderDate) AS EarliestOrder FROM Orders;
-- Result: '2023-01-10'
```

```
-- 3. Grouping by categories
SELECT
    Category,
    MIN(Price) AS MinCategoryPrice
FROM Products
GROUP BY Category;
```

```
-- 4. Minimum string (lexicographic comparison)
SELECT MIN(ProductName) AS FirstProductName FROM Products;
-- Result: 'Milk' (first alphabetically)
```

```
-- 5. Special cases
CREATE TABLE TestValues (Value INT);
INSERT INTO TestValues VALUES (10), (NULL), (5);
SELECT MIN(Value) FROM TestValues; -- 5 (NULL ignored)
```

PI

The function returns the value of the mathematical constant π (pi).

Main capabilities:

- Returns the constant $\pi \approx 3.1415926536$
- Precision - 10 decimal places
- Requires no parameters
- Returns a value of type float

Input parameters:

- No parameters

Output parameters:

- Value of π (type float)

Typical usage scenarios:

- Calculating circumference and area of a circle
- Trigonometric calculations
- Geometric calculations (volumes, surfaces)
- Graphical transformations and animations
- Engineering calculations (mechanics, physics)

Query examples:

```
-- 1. Getting the value of  $\pi$ 
SELECT PI() AS PiValue;
-- Result: 3.141592653589793
```

```
-- 2. Usage in calculations (circumference)
DECLARE @radius FLOAT = 5;
SELECT 2 * PI() * @radius AS Circumference;
-- Result: 31.41592653589793
```

```
-- 3. Area of a circle
SELECT PI() * POWER(10, 2) AS CircleArea;
-- Result: 314.1592653589793
```

```
-- 4. Trigonometric calculations
SELECT SIN(PI()/2) AS Sin90Deg; -- 1.0
```

POWER

The function returns the result of raising a number to the specified power.

Main capabilities:

- Calculates x^n (x to the power of n)
- Supports integer and fractional powers
- Inverse function - LOG (logarithm)

Input parameters:

- x - base (any real number)
- y - exponent (any real number)

Output parameters:

- Result of exponentiation (type float)
- NULL if any parameter is NULL

Typical usage scenarios:

- Squaring or cubing values (e.g., calculating area or volume)
- Complex financial calculations (compound interest, investment growth)
- Scientific calculations (physical and chemical formulas)
- Unit conversion (e.g., square meters to square centimeters)

- Calculating exponential growth (population, viral activity)

Query examples:

```
-- 1. Raising to an integer power
SELECT POWER(5, 3) AS Result; -- 53
-- Result: 125.0
```

```
-- 2. Raising to a fractional power
SELECT POWER(9, 0.5) AS SquareRoot; -- √9
-- Result: 3.0
```

```
-- 3. Special cases
SELECT
    POWER(5, 0) AS AnyToZero, -- 1.0
    POWER(1, 100) AS OneToAny, -- 1.0
    POWER(0, 5) AS ZeroToPositive; -- 0.0
```

```
-- 4. Using variables
DECLARE @base FLOAT = 2.5, @exp FLOAT = 3;
SELECT POWER(@base, @exp) AS CustomPower;
-- Result: 15.625
```

```
-- 5. Complex calculations
SELECT POWER(10, LOG10(2)) AS InverseLog;
-- Result: 0.2.0
```

RAND

The function returns a pseudo-random number of type float in the range from 0 to 1.

Main capabilities:

- Generates a random number from 0 to 1
- Can accept a seed value for reproducibility
- Returns a value of type float
- With the same seed, generates identical sequences

Input parameters:

- seed - optional integer for initializing the generator

Output parameters:

- Random number ($0 \leq n < 1$) of type float
- NULL if seed is NULL

Typical usage scenarios:

- Generating random numbers in the range from 0 to 1
- Creating pseudo-random values for test data
- Use in mathematical calculations requiring random coefficients
- Simulating probabilistic processes (e.g., draws or random events)

Query examples:

```
-- 1. Basic usage (random number)
SELECT RAND() AS RandomValue;
-- Result: e.g., 0.71359199321292355
```

```
-- 2. With a fixed seed (reproducible result)
SELECT RAND(100) AS SeededRandom;
-- Always: 0.71543665736748563
```

```
-- 3. Generating numbers in a given range (from 10 to 20)
SELECT 10 + (RAND() * 10) AS RandomInRange;
-- Result: e.g., 15.327590423332144
```

```
-- 4. Usage in INSERT
CREATE TABLE #RandomNumbers (ID INT IDENTITY, Value FLOAT);
INSERT INTO #RandomNumbers (Value)
SELECT RAND() FROM (VALUES (1),(2),(3),(4),(5)) AS nums(n);
SELECT * FROM #RandomNumbers;
```

ROUND

The function rounds a number to the specified precision.

Main capabilities:

- Rounds numbers to a given number of decimal places
- Supports different data types (decimal, float, money)
- Can round up or down
- Allows specifying the rounding type (default - to nearest)

Input parameters:

- numeric_expression - number to round
- length - number of decimal places
- function - optional rounding parameter:
- 0 or omitted - standard rounding (default)
- 1 - truncation (similar to TRUNCATE)

Output parameters:

- Rounded value of the same type as the input
- NULL if the input value is NULL

Typical usage scenarios:

- Rounding numeric values to a specified number of decimal places
- Financial calculations requiring precision to a certain digit (cents, kopecks, etc.)
- Simplifying fractional numbers for report readability
- Processing results of mathematical operations before further calculations
- Converting data to a unified format before saving or displaying

Query examples:

```
-- 1. Standard rounding (to 2 places)
SELECT ROUND(5.323232, 2) AS RoundedValue;
-- Result: 5.32
```

```
-- 2. Rounding to integer  
SELECT ROUND(5.678, 0) AS IntegerRound;  
-- Result: 6
```

```
-- 3. Truncation (without rounding)  
SELECT ROUND(5.678, 2, 1) AS TruncatedValue;  
-- Result: 5.67
```

```
-- 4. Rounding monetary values  
SELECT ROUND($123.4567, 2) AS MoneyRounded;  
-- Result: 123.46
```

```
-- 5. Negative precision (rounding to tens)  
SELECT ROUND(123.456, -1) AS RoundedToTens;  
-- Result: 120
```

SIGN

The function returns the sign of the specified numeric expression.

Main capabilities:

- Determines the sign of a number (-1, 0, or +1)
- Works with all numeric data types
- Returns an integer value
- Does not change the number itself, only determines its sign

Input parameters:

- `numeric_expression` - numeric expression to analyze

Output parameters:

- -1 if the number is negative
- 0 if the number is zero
- +1 if the number is positive
- NULL if the input value is NULL

Typical usage scenarios:

- Classifying numbers by sign (positive/negative/zero)
- Conditional data processing depending on the sign of the value
- Filtering records by the sign of numeric values
- Mathematical calculations requiring analysis of a number's sign

Query examples:

```
-- 1. Determining the sign of a number
SELECT SIGN(41) AS SignResult;
-- Result: 1
```

SIN

The function returns the trigonometric sine of the specified angle.

Main capabilities:

- Calculates the sine of an angle specified in radians
- Supports all numeric data types
- Returns a value of type FLOAT
- Can be used in mathematical and engineering calculations

Input parameters:

- angle – angle in radians (numeric expression: INT, DECIMAL, FLOAT, etc.)

Output parameters:

- Sine value in the range from -1 to 1 (type FLOAT)
- NULL if the input value is NULL

Typical usage scenarios:

- Trigonometric calculations in mathematics and physics
- Processing geometric data (coordinates, angles)
- Generating wave signals (data analysis, sound vibrations)
- Calculations in computer graphics and animation

Query examples:

```
-- Sine of angle  $\pi/2$  (90 degrees)
SELECT SIN(PI()/2) AS SinValue;
-- Result: 1
```

```
-- Sine of zero angle
SELECT SIN(0) AS SinZero;
-- Result: 0
```

SQRT

The function calculates the square root of the specified number.

Main capabilities:

- Returns the square root of a given numeric value
- Supports numeric data types (INT, DECIMAL, FLOAT, etc.)
- Returns NULL if the input value is NULL
- Causes an error when a negative number is passed

Input parameters:

- number – number to calculate the square root of (non-negative numeric expression)

Output parameters:

- Square root of the number (type FLOAT)
- NULL if the input value is NULL
- Error if the number is negative

Typical usage scenarios:

- Mathematical and statistical calculations (standard deviation, distance between points)
- Financial calculations (compound interest, volatility)
- Engineering calculations (physics, electrical engineering)

- Geometric measurements (diagonal length, circle radius)
- Checking data correctness (negative values are not allowed)

Query examples:

```
-- Square root of 25  
SELECT SQRT(25) AS Root;  
-- Result: 5
```

```
-- Square root of a fractional number  
SELECT SQRT(2.25) AS Root;  
-- Result: 1.5
```

```
-- Handling NULL values  
SELECT SQRT(NULL) AS Root;  
-- Result: NULL
```

```
-- Error with negative number  
SELECT SQRT(-1) AS Root;
```

SQUARE

The function returns the square of the specified number.

Main capabilities:

- Calculates the value of a number raised to the second power
- Supports all numeric data types (INT, DECIMAL, FLOAT, etc.)
- Returns a value of the same type as the input parameter (except for integer overflows)
- Works with positive, negative numbers, and zero

Input parameters:

- number - numeric expression to square

Output parameters:

- Square of the input number (preserves the type of the input value)

- NULL if the input value is NULL

Typical usage scenarios:

- Mathematical and statistical calculations (variance, standard deviation)
- Physical formulas (kinetic energy, area calculations)
- Financial calculations (compound interest, profitability analysis)
- Geometric calculations (area of a square, Pythagorean theorem)
- Engineering calculations (power, resistance)

Query examples:

```
-- Square of an integer  
SELECT SQUARE(5) AS Result;  
-- Result: 25
```

```
-- Square of a fractional number  
SELECT SQUARE(2.5) AS Result;  
-- Result: 6.25
```

```
-- Square of a negative number  
SELECT SQUARE(-3) AS Result;  
-- Result: 9
```

```
-- Handling NULL values  
SELECT SQUARE(NULL) AS Result;  
-- Result: NULL
```

STDEV

The function calculates the statistical standard deviation for the specified set of values.

Main capabilities:

- Calculates standard deviation as a measure of data dispersion
- Ignores NULL values in the sample
- Supports numeric data types (INT, DECIMAL, FLOAT, etc.)

Input parameters:

- expression — numeric expression or column name with data for analysis

Output parameters:

- Standard deviation (type FLOAT)
- NULL if there is insufficient data in the sample (less than 2 rows)

Typical usage scenarios:

- Analyzing volatility of financial indicators
- Quality control in production
- Processing experimental data
- Statistical sample analysis

Query examples:

```
-- Creating and populating a products table
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName NVARCHAR(100),
    Price DECIMAL(10, 2)
);
INSERT INTO Products VALUES
(1, 'Milk', 50.00),
(2, 'Bread', 30.00),
(3, 'Cheese', 220.00),
(4, 'Eggs', 70.00),
(5, 'Meat', 350.00);
```

```
-- Calculating standard deviation of prices
SELECT STDEV(Price) AS PriceDeviation FROM Products;
```

```
-- Creating and populating a students table
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    GroupName NVARCHAR(20),
    ExamScore INT
);
INSERT INTO Students VALUES
(1, 'Group A', 85),
(2, 'Group A', 90),
(3, 'Group A', 78),
(4, 'Group B', 92),
(5, 'Group B', 88),
(6, 'Group B', 95);
```

```
-- Comparing standard deviations of scores between groups
SELECT
    GroupName,
    AVG(ExamScore) AS AvgScore,
    STDEV(ExamScore) AS ScoreDeviation
FROM Students
GROUP BY GroupName;
```

STDEVP

The function calculates the standard deviation for the population.

Main capabilities:

- Returns the standard deviation for the entire population of data
- Ignores NULL values in calculations
- Requires at least 1 value for calculation (unlike STDEV, which needs 2 values)
- Supports numeric data types (INT, DECIMAL, FLOAT, etc.)

Input parameters:

- expression - numeric expression or column

Output parameters:

- Population standard deviation (FLOAT)
- NULL if all values are NULL

Typical usage scenarios:

- Analyzing data for the entire population (not a sample)
- Quality control during full product inspection
- Statistical analysis of complete data sets

Query examples:

```
-- Creating and populating an employees table
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Department NVARCHAR(50),
    Salary DECIMAL(10, 2)
);
INSERT INTO Employees VALUES
(1, 'IT', 85000.00),
(2, 'IT', 92000.00),
(3, 'HR', 65000.00),
(4, 'HR', 72000.00),
(5, 'Sales', 58000.00),
(6, 'Sales', NULL);
```

```
-- Standard deviation of salaries across the entire company
SELECT STDEVP(Salary) AS SalaryDeviation FROM Employees;
```

```
-- Comparing departments
SELECT
    Department,
    AVG(Salary) AS AvgSalary,
    STDEVP(Salary) AS SalaryDeviation
FROM Employees
WHERE Salary IS NOT NULL
GROUP BY Department;
```

```
-- Creating and populating a measurements table
CREATE TABLE QualityMeasurements (
    MeasurementID INT PRIMARY KEY,
    ProductCode NVARCHAR(20),
    Value DECIMAL(8, 3)
);
INSERT INTO QualityMeasurements VALUES
(1, 'A-100', 10.254),
(2, 'A-100', 10.312),
(3, 'A-100', 10.198),
(4, 'B-200', 15.687),
(5, 'B-200', 15.723),
(6, 'B-200', 15.655);
```

```
-- Analyzing production precision by product
SELECT
    ProductCode,
    STDEVP(Value) AS ProductionPrecision
FROM QualityMeasurements
```

SUM

The function returns the sum of all values in the specified column or expression.

Main capabilities:

- Calculates the sum of values in a column or expression
- Ignores NULL values in calculations
- Supports numeric data types (INT, DECIMAL, FLOAT, MONEY, etc.)
- Can be used with GROUP BY for data aggregation by groups
- Works with DISTINCT for summing unique values

Input parameters:

- expression - numeric expression or column to sum
- DISTINCT (optional) - flag to sum only unique values

Output parameters:

- Sum of values (preserves the data type of the input data)

- NULL if all values are NULL or the table is empty

Typical usage scenarios:

- Financial reports (total revenue, sum of sales)
- Resource accounting (total quantity of goods in stock)
- Performance analysis (total working time)
- Statistical calculations (in combination with other aggregate functions)

Query examples:

```
-- Creating and populating a sales table
CREATE TABLE Sales (
    SaleID INT PRIMARY KEY,
    ProductID INT,
    SaleDate DATE,
    Amount DECIMAL(10, 2),
    Quantity INT
);
INSERT INTO Sales VALUES
(1, 101, '2023-01-15', 150.50, 2),
(2, 102, '2023-01-15', 200.00, 1),
(3, 101, '2023-01-16', 75.25, 1),
(4, 103, '2023-01-17', 300.75, 3),
(5, 102, '2023-01-18', 400.00, 2),
(6, 101, '2023-01-19', 225.75, 3);
```

```
-- 1. Total sum of all sales
SELECT SUM(Amount) AS TotalSales FROM Sales;
```

```
-- 2. Sum of sales by product
SELECT
    ProductID,
    SUM(Amount) AS ProductTotal
FROM Sales
GROUP BY ProductID
ORDER BY ProductTotal DESC;
```

```
-- 3. Total quantity of goods sold
SELECT SUM(Quantity) AS TotalQuantity FROM Sales;
```

```
-- 4. Using DISTINCT (sum of unique values)
SELECT SUM(DISTINCT Amount) AS UniqueAmountsSum FROM Sales;
```

```
-- 5. Sum with WHERE condition
SELECT SUM(Amount) AS WeekendSales
FROM Sales
WHERE DATEPART(WEEKDAY, SaleDate) IN (1, 7);
```

```
-- 6. Combining with other functions
SELECT
    AVG(Amount) AS AvgSale,
    SUM(Amount) AS TotalSales,
    COUNT(*) AS SalesCount
FROM Sales;
```

TAN

The function returns the tangent of the specified angle.

Main capabilities:

- Calculates the trigonometric tangent of an angle specified in radians
- Supports numeric data types (INT, DECIMAL, FLOAT, etc.)
- Returns a value of type FLOAT
- Can be used in complex mathematical calculations

Input parameters:

- angle – angle in radians (numeric expression)

Output parameters:

- Tangent of the angle (FLOAT)
- NULL if the input value is NULL

Typical usage scenarios:

- Trigonometric calculations in mathematics and physics
- Engineering calculations (mechanics, electrical engineering)

- Computer graphics and geometric transformations
- Scientific research and data analysis

Query examples:

```
-- Tangent of angle  $\pi/4$  (45 degrees)
SELECT TAN(PI()/4) AS TangentValue;
-- Result: 1.0
```

```
-- Tangent of zero angle
SELECT TAN(0) AS ZeroTangent;
-- Result: 0
```

```
-- Tangent of 30 degrees (converted to radians first)
SELECT TAN(RADIANS(30)) AS Tan30Degrees;
-- Result: 0.577
```

```
-- Usage in mathematical expressions
SELECT TAN(angle) / (1 - TAN(angle)) AS TrigResult FROM Calculations;
```

```
-- Handling NULL values
SELECT TAN(NULL) AS NullResult;
-- Result: NULL
```

VAR

The function calculates the sample variance (unbiased estimator of the population variance).

Main capabilities:

- Calculates a measure of data dispersion in a sample
- Uses the formula with (n-1) in the denominator
- Ignores NULL values
- Requires at least 2 values for calculation
- Supports numeric data types (INT, DECIMAL, FLOAT, etc.)

Input parameters:

- column_name - numeric column for analysis

Output parameters:

- Variance value (FLOAT)
- NULL if there are fewer than 2 values or all values are NULL

Typical usage scenarios:

- Statistical data analysis
- Production quality control
- Financial risk analysis
- Scientific research
- Analysis of business performance indicators

Query examples:

```
-- Example 1: Analyzing product prices
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    Category VARCHAR(50),
    Price DECIMAL(10,2)
);
INSERT INTO Products VALUES
(1, 'Electronics', 1200.00),
(2, 'Electronics', 950.50),
(3, 'Electronics', 1100.00),
(4, 'Clothing', 45.99),
(5, 'Clothing', 55.50),
(6, 'Clothing', 60.00),
(7, 'Furniture', 450.00),
(8, 'Furniture', 1200.00),
(9, 'Furniture', NULL);
```

```
-- Variance of prices by category
SELECT
    Category,
    AVG(Price) AS AvgPrice,
    VAR(Price) AS PriceVariance
FROM Products
WHERE Price IS NOT NULL
GROUP BY Category;
```

```
-- Example 2: Analyzing academic performance
CREATE TABLE StudentGrades (
    StudentID INT,
    Subject VARCHAR(50),
    Grade INT,
    Semester INT
);
INSERT INTO StudentGrades VALUES
(1, 'Math', 85, 1),
(1, 'Math', 90, 2),
(2, 'Math', 78, 1),
(2, 'Math', 82, 2),
(3, 'Physics', 92, 1),
(3, 'Physics', 88, 2),
(4, 'Physics', 75, 1),
(4, 'Physics', 80, 2),
(5, 'Chemistry', 90, 1),
(5, 'Chemistry', 85, 2);
```

```
-- Variance of grades by subject
SELECT
    Subject,
    VAR(Grade) AS GradeVariance,
    SQRT(VAR(Grade)) AS StandardDeviation
FROM StudentGrades
GROUP BY Subject;
```

```
-- Example 3: Analyzing sales
CREATE TABLE DailySales (
    SaleDate DATE,
    Amount DECIMAL(12,2),
    Region VARCHAR(50)
);
INSERT INTO DailySales VALUES
('2023-01-01', 1200.00, 'North'),
('2023-01-02', 950.50, 'North'),
('2023-01-03', 1100.00, 'North'),
('2023-01-01', 750.00, 'South'),
('2023-01-02', 820.00, 'South'),
('2023-01-03', 780.00, 'South'),
('2023-01-01', 1500.00, 'East'),
('2023-01-02', NULL, 'East'),
('2023-01-03', 1400.00, 'East');
```

```
-- Variance of sales by region
SELECT
    Region,
    VAR(Amount) AS SalesVariance
FROM DailySales
WHERE Amount IS NOT NULL
GROUP BY Region;
```

VARP

The function calculates the variance for the population.

Main capabilities:

- Calculates a measure of data dispersion for the entire population
- Uses the formula with N in the denominator (biased estimator)
- Ignores NULL values in calculations
- Requires at least 1 value for calculation
- Supports numeric data types (INT, DECIMAL, FLOAT, etc.)

Input parameters:

- column_name - numeric column for analysis

Output parameters:

- Population variance (FLOAT)
- NULL if all values are NULL

Typical usage scenarios:

- Analyzing data for the entire population
- Quality control during full product inspection
- Statistical analysis of complete data sets
- Financial analysis of full portfolio data
- Estimating characteristics of the entire production cycle

Query examples:

```
-- Creating an IT department employee table
CREATE TABLE ITDepartment (
    EmployeeID INT PRIMARY KEY,
    Position VARCHAR(50),
    Salary DECIMAL(10,2),
    ExperienceYears INT
);
INSERT INTO ITDepartment VALUES
(1, 'Junior Developer', 50000.00, 1),
(2, 'Developer', 75000.00, 3),
(3, 'Senior Developer', 95000.00, 5),
(4, 'Team Lead', 110000.00, 7),
(5, 'Architect', 125000.00, 10),
(6, 'Manager', 140000.00, 8),
(7, 'Director', 180000.00, 12);
```

```
-- Variance of salaries across the entire department
SELECT
    VARP(Salary) AS SalaryVariance,
    SQRT(VARP(Salary)) AS SalaryStdDev
FROM ITDepartment;
```

```
-- Creating a product measurements table
CREATE TABLE ProductMeasurements (
    BatchID INT,
    ProductID INT,
    Measurement DECIMAL(8,3),
    QualityCheck BIT
);
INSERT INTO ProductMeasurements VALUES
(1, 1001, 10.251, 1),
(1, 1002, 10.248, 1),
(1, 1003, 10.254, 1),
(2, 1001, 10.362, 0),
(2, 1002, 10.358, 0),
(2, 1003, 10.355, 0),
(3, 1001, 10.201, 1),
(3, 1002, 10.198, 1),
(3, 1003, NULL, NULL);
```

```
-- Variance of measurements by batch
SELECT
    BatchID,
    AVG(Measurement) AS AvgMeasurement,
    VARP(Measurement) AS MeasurementVariance
FROM ProductMeasurements
WHERE Measurement IS NOT NULL
GROUP BY BatchID;
```

```
-- Creating a branch sales table
CREATE TABLE BranchSales (
    BranchID INT,
    Year INT,
    Quarter INT,
    SalesAmount DECIMAL(12,2)
);
INSERT INTO BranchSales VALUES
(1, 2023, 1, 1250000.00),
(1, 2023, 2, 1350000.00),
(1, 2023, 3, 1420000.00),
(1, 2023, 4, 1500000.00),
(2, 2023, 1, 980000.00),
(2, 2023, 2, 1050000.00),
(2, 2023, 3, 1100000.00),
(2, 2023, 4, 1150000.00),
(3, 2023, 1, 750000.00),
(3, 2023, 2, 820000.00),
(3, 2023, 3, NULL),
(3, 2023, 4, 900000.00);
```

```
-- Variance of sales by year (all branches)
SELECT
    Year,
    VARP(SalesAmount) AS SalesVariance
FROM BranchSales
WHERE SalesAmount IS NOT NULL
GROUP BY Year;
```

METADATA

APP_NAME

The function returns the name of the client application that established the current connection.

Main capabilities:

- Returns the application name passed during connection establishment
- Supports string data type (NVARCHAR)
- Can be used for connection auditing and control
- Allows identifying the source of queries

Input parameters:

- No parameters

Output parameters:

- Application name (NVARCHAR)
- NULL if the application name was not specified during connection

Typical usage scenarios:

- Auditing and monitoring database connections
- Diagnosing connection problems
- Analyzing load from different client applications

Query examples:

```
-- Getting the name of the current application
SELECT APP_NAME() AS CurrentApplication;
```

COL_LENGTH

The function returns the length of a column in bytes.

Main capabilities:

- Returns the length of a column in a table or view
- Supports all data types

Input parameters:

- table_name - name of the table or view (NVARCHAR)
- column_name - name of the column (NVARCHAR)

Output parameters:

- Column length in bytes (INT)
- NULL if the table or column does not exist

Typical usage scenarios:

- Checking table structure
- Query optimization based on column sizes

Query examples:

```
-- Create a test table
CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    HireDate DATE,
    Salary DECIMAL(10,2),
    Photo VARBINARY(MAX)
);
```

```
-- 1. Getting column lengths
SELECT
    COL_LENGTH('Employee', 'EmployeeID') AS ID_Length,
    COL_LENGTH('Employee', 'FirstName') AS FirstName_Length,
    COL_LENGTH('Employee', 'Salary') AS Salary_Length,
    COL_LENGTH('Employee', 'Photo') AS Photo_Length;
```

```
-- 2. Analyzing all columns of a table
SELECT
    c.name AS ColumnName,
    tp.name AS DataType,
    c.max_length AS MaxLength,
    COL_LENGTH('Employee', c.name) AS ActualLength
FROM sys.columns c
JOIN sys.types tp ON c.user_type_id = tp.user_type_id
WHERE c.object_id = OBJECT_ID('Employee');
```

```
-- 3. Checking column existence before query
IF COL_LENGTH('Employee', 'MiddleName') IS NOT NULL
BEGIN
    SELECT EmployeeID, MiddleName FROM Employee;
END
ELSE
BEGIN
    SELECT EmployeeID, FirstName, LastName FROM Employee;
END;
```

```
-- 4. Storage optimization
SELECT
    t.name AS TableName,
    c.name AS ColumnName,
    COL_LENGTH(t.name, c.name) AS ColumnLength
FROM sys.tables t
JOIN sys.columns c ON t.object_id = c.object_id
ORDER BY COL_LENGTH(t.name, c.name) DESC;
```

COL_NAME

The function returns the name of a table column by its identifier.

Main capabilities:

- Returns the column name by table ID and column ID
- Works with tables and views
- Supports all database objects
- Allows getting column names without direct access to system views

Input parameters:

- table_id - table identifier (INT)
- column_id - column identifier (INT)

Output parameters:

- Column name (NVARCHAR(128))
- NULL if the table or column does not exist

Typical usage scenarios:

- Getting metadata about table structure
- Analyzing database schema
- Creating reports on object structure
- Checking column existence
- Auditing database structure changes

Query examples:

```
-- Create a test table
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Email NVARCHAR(100),
    RegistrationDate DATETIME
);
```

```
-- 1. Getting column names by their IDs
SELECT
    COL_NAME(OBJECT_ID('Customers'), 1) AS Column1, -- CustomerID
    COL_NAME(OBJECT_ID('Customers'), 2) AS Column2, -- FirstName
    COL_NAME(OBJECT_ID('Customers'), 3) AS Column3; -- LastName
```

```
-- 2. Getting all column names of a table
SELECT
    c.column_id AS ColumnID,
    COL_NAME(c.object_id, c.column_id) AS ColumnName,
    t.name AS DataType
FROM sys.columns c
JOIN sys.types t ON c.user_type_id = t.user_type_id
WHERE c.object_id = OBJECT_ID('Customers')
ORDER BY c.column_id;
```

```
-- 3. Checking existence of a specific column
IF COL_NAME(OBJECT_ID('Customers'), 4) IS NOT NULL
BEGIN
    PRINT 'Email column exists in the Customers table';
END;
```

```
-- 4. Comparing structures of two tables
SELECT
    COL_NAME(OBJECT_ID('Customers'), c.column_id) AS CustomerColumn,
    COL_NAME(OBJECT_ID('Employees'), e.column_id) AS EmployeeColumn
FROM sys.columns c
FULL OUTER JOIN sys.columns e ON c.column_id = e.column_id
WHERE c.object_id = OBJECT_ID('Customers')
    OR e.object_id = OBJECT_ID('Employees');
```

```
-- 5. Getting information about the primary key
SELECT COL_NAME(k.parent_object_id, kc.column_id) AS PrimaryKeyColumn
FROM sys.key_constraints k
JOIN sys.index_columns kc ON k.parent_object_id = kc.object_id
                        AND k.unique_index_id = kc.index_id
WHERE k.type = 'PK'
      AND k.parent_object_id = OBJECT_ID('Customers');
```

COLUMNPROPERTY

The function returns information about the properties of a table column.

Main capabilities:

- Checks various column properties (IDENTITY, NULLABLE, COMPUTED, etc.)
- Returns 1 (true), 0 (false), or NULL
- Works with temporary tables and views

Input parameters:

- table_id - table ID (via OBJECT_ID)
- column_name - column name
- property - type of property to check

Output parameters:

- 1 if the property is true
- 0 if the property is false
- NULL if parameters are invalid

Typical usage scenarios:

- Checking column properties before performing operations
- Generating DDL scripts
- Analyzing database structure
- Validating data before migration

Query examples:


```
--1. Checking AllowsNull (allows NULL)
CREATE TABLE ExampleTable (
    ID INT PRIMARY KEY,
    Name VARCHAR(100) NOT NULL,
    Description VARCHAR(200) NULL
);
SELECT
    COLUMNPROPERTY(OBJECT_ID('ExampleTable'), 'Name', 'AllowsNull') AS Name_Nullable,
-- 0
    COLUMNPROPERTY(OBJECT_ID('ExampleTable'), 'Description', 'AllowsNull') AS
Desc_Nullable; -- 1
```

```
--2. Checking IsIdentity (auto-increment)
CREATE TABLE IdentityTest (
    ID INT IDENTITY(1,1),
    Code VARCHAR(10)
);
SELECT
    COLUMNPROPERTY(OBJECT_ID('IdentityTest'), 'ID', 'IsIdentity') AS IsID_Identity, --
1
    COLUMNPROPERTY(OBJECT_ID('IdentityTest'), 'Code', 'IsIdentity') AS
IsCode_Identity; -- 0
```

```
--3. Checking IsComputed (computed column)
CREATE TABLE ComputedTest (
    Quantity INT,
    Total AS (Quantity * 10)
);
SELECT
    COLUMNPROPERTY(OBJECT_ID('ComputedTest'), 'Quantity', 'IsComputed') AS
Qty_Computed, -- 0
    COLUMNPROPERTY(OBJECT_ID('ComputedTest'), 'Total', 'IsComputed') AS
Total_Computed; -- 1
```

DATABASEPROPERTYEX

The function returns the current settings and properties of the specified database.

Main capabilities:

- Returns the value of the specified database property
- Works with all user and system databases

- Returns a string value or NULL

Input parameters:

- database_name - database name
- property - name of the property to check

Output parameters:

- Property value (SQL_VARIANT)
- NULL if the property does not exist

Query examples:

```
-- 1. Checking collation parameters  
SELECT DATABASEPROPERTYEX('AdventureWorks', 'Collation') AS DatabaseCollation;
```

```
-- 2. Checking compatibility level  
SELECT DATABASEPROPERTYEX('AdventureWorks', 'Version') AS CompatibilityLevel;
```

```
-- 3. Checking database status  
SELECT DATABASEPROPERTYEX('AdventureWorks', 'Status') AS DatabaseStatus;
```

```
-- 4. Checking recovery model  
SELECT DATABASEPROPERTYEX('AdventureWorks', 'Recovery') AS RecoveryModel;
```

```
-- 5. Checking database availability  
SELECT DATABASEPROPERTYEX('AdventureWorks', 'IsInStandBy') AS IsStandbyMode;
```

DB_ID

The function returns the identifier of the specified database in SQL Server.

Main capabilities:

- Returns a unique numeric database identifier
- Works with the current and any other databases on the server

- Returns NULL if the database does not exist

Input parameters:

- database_name - database name (optional parameter)

Output parameters:

- Database identifier
- NULL if the database does not exist

Typical usage scenarios:

- Checking database existence before operations
- Getting database ID for system functions and DMVs
- Server administration and monitoring
- Automating maintenance tasks

Query examples:

```
-- 1. Getting the ID of the current database
SELECT DB_ID() AS CurrentDBID;
```

```
-- 2. Getting the ID of the master system database
SELECT DB_ID('master') AS MasterDBID; -- Always returns 1
```

```
-- 3. Checking existence of a user database
SELECT DB_ID('AdventureWorks') AS AdventureWorksID;
```

```
-- 4. Using in conditional logic
IF DB_ID('MyDatabase') IS NOT NULL
    PRINT 'Database exists';
ELSE
    PRINT 'Database not found';
```

NEXT VALUE FOR

The function returns the next value from the specified sequence.

Main capabilities:

- Generates the next sequence value
- Supports both ascending and descending sequences
- Guarantees uniqueness of values within the sequence
- Can be used in INSERT, UPDATE, and SELECT statements
- Supports value caching for performance improvement

Input parameters:

- sequence_name - sequence name (required parameter, type: sysname)

Output parameters:

- Next sequence value

Typical usage scenarios:

- Generating unique identifiers for primary keys
- Creating serial numbers for documents
- Populating columns with unique values
- Alternative to IDENTITY for more flexible value management

Query examples:

```
-- 1. Creating a sequence
CREATE SEQUENCE dbo.OrderNumberSeq
    START WITH 1000
    INCREMENT BY 1;
```

```
-- 2. Getting the next value
SELECT NEXT VALUE FOR dbo.OrderNumberSeq;
-- Result: 1000 (on first call)
```

```
-- 3. Using in INSERT
INSERT INTO dbo.Orders (OrderID, CustomerID, OrderDate)
VALUES (NEXT VALUE FOR dbo.OrderNumberSeq, 123, GETDATE());
```

```
-- 4. Using multiple values in one statement
SELECT
    NEXT VALUE FOR dbo.OrderNumberSeq AS FirstValue,
    NEXT VALUE FOR dbo.OrderNumberSeq AS SecondValue;
```

OBJECT_DEFINITION

The function returns the definition text of the specified database object.

Main capabilities:

- Returns the source code of SQL objects (procedures, functions, triggers, views)
- Supports both user and system objects
- Returns NULL for encrypted objects
- Outputs text in Unicode format (NVARCHAR)
- Allows analyzing the structure of database objects

Input parameters:

- object_id - object identifier in the database

Output parameters:

- Object definition text
- NULL if the object does not exist, is encrypted, or no access rights

Typical usage scenarios:

- Analyzing and documenting database object code
- Recovering lost definitions
- Comparing different versions of objects
- Debugging and modifying existing code
- Automating documentation processes

Query examples:

```
-- 1. Creating and getting a regular procedure
CREATE PROCEDURE dbo.ExampleProc
AS
BEGIN
    SELECT GETDATE() AS CurrentDate;
END;
SELECT OBJECT_DEFINITION(OBJECT_ID('dbo.ExampleProc'));
-- Result: text CREATE PROCEDURE...
```

```
-- 2. Creating and getting a function
CREATE FUNCTION dbo.CalculateTax(@amount DECIMAL(18,2))
RETURNS DECIMAL(18,2)
AS
BEGIN
    RETURN @amount * 0.2;
END;
```

SELECT OBJECT_DEFINITION(OBJECT_ID('dbo.CalculateTax')); — Result: text CREATE FUNCTION...

OBJECT_ID

The function returns the identifier of the specified object in the database.

Main capabilities:

- Returns the numeric identifier of an object by its name
- Supports all types of database objects (tables, procedures, functions, etc.)
- Returns NULL if the object does not exist
- Works with objects in different schemas
- Allows checking object existence

Input parameters:

- object_name - object name
- object_type - optional object type parameter

Output parameters:

- Object identifier (INT)

- NULL if the object does not exist

Typical usage scenarios:

- Checking object existence before performing operations
- Getting object ID for other system functions

Query examples:

```
with object creation:
-- 1. Creating and checking a table
CREATE TABLE dbo.Employees (
    EmployeeID INT PRIMARY KEY,
    Name NVARCHAR(100),
    Position NVARCHAR(50)
);
SELECT OBJECT_ID('dbo.Employees') AS TableID;
-- Result: numeric table ID (e.g., 123456)
```

```
-- 2. Creating and checking a stored procedure
CREATE PROCEDURE dbo.GetEmployeeByID
    @EmployeeID INT
AS
BEGIN
    SELECT * FROM dbo.Employees WHERE EmployeeID = @EmployeeID;
END;
```

`SELECT OBJECT_ID('dbo.GetEmployeeByID', 'P') AS ProcedureID;`— Result: numeric procedure ID

```
-- 3. Creating and checking a function
CREATE FUNCTION dbo.CalculateBonus(
    @Salary DECIMAL(18,2),
    @Experience INT
)
RETURNS DECIMAL(18,2)
AS
BEGIN
    RETURN @Salary * 0.1 * @Experience;
END;
SELECT OBJECT_ID('dbo.CalculateBonus', 'FN') AS FunctionID;
-- Result: numeric function ID
```

OBJECT_NAME

The function returns the name of a database object by its identifier.

Main capabilities:

- Returns the object name by its numeric identifier
- Supports all types of database objects (tables, procedures, functions, etc.)
- Returns NULL if an object with the specified ID does not exist
- Works with objects in different schemas
- Often used together with OBJECT_ID

Input parameters:

- object_id - numeric object identifier
- database_id - optional database ID parameter

Output parameters:

- Object name (NVARCHAR(128))
- NULL if the object does not exist

Typical usage scenarios:

- Getting object name when working with system views
- Logging and debugging operations with objects
- Recovering object name from a known ID
- Analyzing database metadata

Query examples:


```
with object creation:
-- 1. Creating a table and getting its name by ID
CREATE TABLE dbo.Products (
    ProductID INT PRIMARY KEY,
    ProductName NVARCHAR(100),
    Price DECIMAL(18,2)
);
DECLARE @TableID INT = OBJECT_ID('dbo.Products');
SELECT OBJECT_NAME(@TableID) AS TableName;
-- Result: 'Products'
```

```
-- 2. Creating a procedure and checking
CREATE PROCEDURE dbo.UpdateProductPrice
    @ProductID INT,
    @NewPrice DECIMAL(18,2)
AS
BEGIN
    UPDATE dbo.Products SET Price = @NewPrice WHERE ProductID = @ProductID;
END;
DECLARE @ProcID INT = OBJECT_ID('dbo.UpdateProductPrice');
SELECT OBJECT_NAME(@ProcID) AS ProcedureName;
-- Result: 'UpdateProductPrice'
```

```
-- 3. Creating a function and getting the name
CREATE FUNCTION dbo.GetDiscountedPrice(
    @Price DECIMAL(18,2),
    @Discount DECIMAL(5,2)
)
RETURNS DECIMAL(18,2)
AS
BEGIN
    RETURN @Price * (1 - @Discount/100);
END;
SELECT OBJECT_NAME(OBJECT_ID('dbo.GetDiscountedPrice')) AS FunctionName;
-- Result: 'GetDiscountedPrice'
```

OBJECT_SCHEMA_NAME

The function returns the schema name for the specified database object.

Main capabilities:

- Returns the schema name by object identifier

- Supports all types of database objects (tables, procedures, functions, etc.)
- Returns NULL if an object with the specified ID does not exist
- Works with objects in the current or specified database
- Often used together with OBJECT_NAME and OBJECT_ID

Input parameters:

- object_id - numeric object identifier
- database_id - optional database ID parameter

Output parameters:

- Schema name (NVARCHAR(128))
- NULL if the object does not exist

Typical usage scenarios:

- Getting the fully qualified name of an object
- Analyzing database structure
- Logging operations with objects

Query examples:

```
-- 1. Creating a table in an explicit schema  
CREATE SCHEMA Sales;
```

```
CREATE TABLE Sales.Orders ( OrderID INT PRIMARY KEY, OrderDate DATE,  
CustomerID INT );— Getting the table schema  
DECLARE @TableID INT = OBJECT_ID('Sales.Orders');  
SELECT OBJECT_SCHEMA_NAME(@TableID) AS  
SchemaName;— Result: 'Sales'
```

```
-- 2. Creating a procedure in the dbo schema
CREATE PROCEDURE dbo.GetRecentOrders
AS
BEGIN
    SELECT TOP 10 * FROM Sales.Orders ORDER BY OrderDate DESC;
END;
-- Getting the procedure schema
SELECT OBJECT_SCHEMA_NAME(OBJECT_ID('dbo.GetRecentOrders')) AS ProcSchema;
-- Result: 'dbo'
```

```
-- 3. Creating a function in a custom schema
CREATE SCHEMA Finance;
GO
CREATE FUNCTION Finance.CalculateTax(@Amount DECIMAL(18,2))
RETURNS DECIMAL(18,2)
AS
BEGIN
    RETURN @Amount * 0.2;
END;
-- Getting schema information
SELECT
    OBJECT_NAME(OBJECT_ID('Finance.CalculateTax')) AS FunctionName,
    OBJECT_SCHEMA_NAME(OBJECT_ID('Finance.CalculateTax')) AS FunctionSchema;
-- Result: 'CalculateTax', 'Finance'
```

PARSENAME

The function breaks down an object name into parts and returns the specified component.

Main capabilities:

- Parses compound object names
- Returns server, database, schema, or object name
- Supports 4-component naming notation
- Does not check object existence

Input parameters:

- object_name - full object name
- object_part - number of the requested part (INT: 1-4)

Output parameters:

- Requested part of the name (NVARCHAR)
- NULL if the part is missing

Typical usage scenarios:

- Analyzing full object names
- Parsing connection strings
- Processing metadata

Query examples:

```
-- 1. CREATING TEST OBJECTS IN THE CURRENT DATABASE
-- Create schema if it doesn't exist
IF NOT EXISTS (SELECT * FROM sys.schemas WHERE name = 'Sales')
BEGIN
    EXEC('CREATE SCHEMA Sales');
END
GO
```

```
-- Create a test table
IF OBJECT_ID('Sales.Orders', 'U') IS NULL
BEGIN
    CREATE TABLE Sales.Orders (
        OrderID INT PRIMARY KEY,
        CustomerName NVARCHAR(100)
    );
END
GO
```

```
-- Create a test procedure (correct way)
IF OBJECT_ID('dbo.GetOrders', 'P') IS NULL
BEGIN
    EXEC('
    CREATE PROCEDURE dbo.GetOrders
    AS
    BEGIN
        SELECT * FROM Sales.Orders;
    END
    ');
END
GO
```

```
-- 2. BASIC EXAMPLES OF NAME PARSING
-- Parsing a full table name
SELECT
    PARSENAME('Sales.Orders', 1) AS ObjectName,    -- 'Orders'
    PARSENAME('Sales.Orders', 2) AS SchemaName;    -- 'Sales'
-- Parsing a full procedure name
SELECT
    PARSENAME('dbo.GetOrders', 1) AS ProcedureName, -- 'GetOrders'
    PARSENAME('dbo.GetOrders', 2) AS SchemaName;    -- 'dbo'
```

SCHEMA_ID

The function returns the identifier of a schema by its name.

Main capabilities:

- Returns the ID of a database schema
- Supports both user and system schemas
- Returns NULL if the schema does not exist
- Works with schema names in any case

Input parameters:

- schema_name - schema name

Output parameters:

- Schema identifier (INT)

- NULL if the schema does not exist

Typical usage scenarios:

- Checking schema existence
- Getting schema ID for system functions
- Auditing database structure

Query examples:

```
-- 1. Getting IDs of standard schemas
SELECT
    SCHEMA_ID('dbo') AS DboSchemaID,
    SCHEMA_ID('sys') AS SysSchemaID;
```

```
-- 2. Creating and checking a user schema
IF SCHEMA_ID('Sales') IS NULL
BEGIN
    EXEC('CREATE SCHEMA Sales');
END
SELECT SCHEMA_ID('Sales') AS SalesSchemaID;
```

```
-- 3. Checking a non-existent schema
SELECT SCHEMA_ID('NonExistingSchema') AS NonExistingSchemaID;
-- Result: NULL
```

```
-- 4. Usage in system views
SELECT
    s.name AS SchemaName,
    s.schema_id AS DirectID,
    SCHEMA_ID(s.name) AS FunctionID
FROM sys.schemas s
WHERE s.name IN ('dbo', 'sys', 'Sales');
```

SCHEMA_NAME

The function returns the name of a schema by its identifier.

Main capabilities:

- Returns the schema name by its ID
- Supports both user and system schemas
- Returns NULL if a schema with the specified ID does not exist
- Works with schema identifiers in the current database

Input parameters:

- schema_id - numeric schema identifier (INT)

Output parameters:

- Schema name
- NULL if the schema does not exist

Typical usage scenarios:

- Getting schema name when working with system views
- Recovering schema name from a known ID
- Analyzing database metadata
- Forming full object names

Query examples:

```
-- 1. First, get the IDs of the main schemas
DECLARE @DboSchemaID INT = SCHEMA_ID('dbo');
DECLARE @SysSchemaID INT = SCHEMA_ID('sys');
DECLARE @GuestSchemaID INT = SCHEMA_ID('guest');
-- Check that the schemas exist
IF @DboSchemaID IS NULL
    PRINT 'dbo schema not found!';
ELSE
    PRINT 'dbo schema ID: ' + CAST(@DboSchemaID AS VARCHAR);
IF @SysSchemaID IS NULL
    PRINT 'sys schema not found!';
ELSE
    PRINT 'sys schema ID: ' + CAST(@SysSchemaID AS VARCHAR);
```

```
-- 2. Getting a list of objects in dbo and sys
SELECT
    *.name AS ObjectName,
    *.type_desc AS ObjectType,
    CASE *.schema_id
        WHEN @DboSchemaID THEN 'dbo'
        WHEN @SysSchemaID THEN 'sys'
        ELSE SCHEMA_NAME(*.schema_id)
    END AS SchemaName
FROM sys.objects *
WHERE *.schema_id IN (@DboSchemaID, @SysSchemaID)
ORDER BY SchemaName, ObjectName;
```

```
-- 3. Checking existence of a temporary schema
IF OBJECT_ID('tempdb..#TempTable') IS NOT NULL
BEGIN
    DECLARE @TempSchemaID INT = SCHEMA_ID('tempdb');
    SELECT
        @TempSchemaID AS TempSchemaID,
        SCHEMA_NAME(@TempSchemaID) AS TempSchemaName;
END
```

```
-- 4. Comparing IDs of standard schemas
SELECT
    SCHEMA_NAME(@DboSchemaID) AS DboSchema,
    SCHEMA_NAME(@SysSchemaID) AS SysSchema,
    CASE
        WHEN @DboSchemaID < @SysSchemaID THEN 'dbo has a smaller ID'
        ELSE 'sys has a smaller or equal ID'
    END AS ComparisonResult;
```

SCOPE_IDENTITY

The function returns the last identity value inserted into a table in the current scope.

Main capabilities:

- Returns the last IDENTITY value generated in the current session and scope
- Works only with IDENTITY columns
- Returns NULL if there were no insert operations
- Independent of triggers that may run after insertion

Input parameters:

- No parameters

Output parameters:

- Last IDENTITY value (type depends on the column)
- NULL if there were no insert operations or they did not affect IDENTITY columns

Typical usage scenarios:

- Getting the ID of a just inserted record
- Linking child records to parent records
- Logging insert operations
- Processing chains of related inserts

Query examples:

```
-- 1. Create test tables with IDENTITY
IF OBJECT_ID('dbo.Customers', 'U') IS NULL
BEGIN
    CREATE TABLE dbo.Customers (
        CustomerID INT IDENTITY(1,1) PRIMARY KEY,
        CustomerName NVARCHAR(100) NOT NULL,
        Email NVARCHAR(100)
    );
```

```
CREATE TABLE dbo.Orders (
    OrderID INT IDENTITY(100,1) PRIMARY KEY,
    CustomerID INT NOT NULL,
    OrderDate DATETIME DEFAULT GETDATE(),
    Amount DECIMAL(10,2),
    FOREIGN KEY (CustomerID) REFERENCES dbo.Customers(CustomerID)
);
END
```

```
-- 2. Basic insert example
INSERT INTO dbo.Customers (CustomerName, Email)
VALUES ('Ivan Petrov', 'ivan@example.com');
DECLARE @NewCustomerID INT = SCOPE_IDENTITY();
IF @NewCustomerID IS NOT NULL
BEGIN
    PRINT 'New customer added with ID: ' + CAST(@NewCustomerID AS VARCHAR);
```

```
-- Insert an order for this customer
INSERT INTO dbo.Orders (CustomerID, Amount)
VALUES (@NewCustomerID, 1500.00);
```

```
    PRINT 'Order added for customer ' + CAST(@NewCustomerID AS VARCHAR);
END
ELSE
    PRINT 'Failed to get ID of new customer';
```

```
-- 3. Example with multiple inserts check
BEGIN TRANSACTION;
    -- Insert several customers
    INSERT INTO dbo.Customers (CustomerName) VALUES ('Anna Sidorova');
    INSERT INTO dbo.Customers (CustomerName) VALUES ('Petr Ivanov');
    -- Get ID of only the last customer
    DECLARE @LastCustomerID INT = SCOPE_IDENTITY();
    IF @LastCustomerID IS NOT NULL
    BEGIN
        PRINT 'Last added customer: ' + CAST(@LastCustomerID AS VARCHAR);
        -- Insert an order for the last customer
        INSERT INTO dbo.Orders (CustomerID, Amount)
        VALUES (@LastCustomerID, 2300.50);
    END
COMMIT TRANSACTION;
```

```
-- 4. Comparison with @@IDENTITY and IDENT_CURRENT
-- Create a trigger that also inserts records
IF OBJECT_ID('dbo.trg_AfterInsertOrder', 'TR') IS NULL
BEGIN
    EXEC('
    CREATE TRIGGER dbo.trg_AfterInsertOrder
    ON dbo.Orders
    AFTER INSERT
    AS
    BEGIN
        -- Trigger inserts a record into the log
        IF OBJECT_ID(''dbo.OrderLog'', ''U'') IS NULL
            CREATE TABLE dbo.OrderLog (LogID INT IDENTITY(1,1), OrderID INT, LogDate
DATETIME);
        INSERT INTO dbo.OrderLog (OrderID, LogDate)
        SELECT OrderID, GETDATE() FROM inserted;
    END
    ');
END
-- Insert data and compare functions
INSERT INTO dbo.Customers (CustomerName) VALUES ('Sergey Vasilyev');
DECLARE @CustomerID INT = SCOPE_IDENTITY();
IF @CustomerID IS NOT NULL
BEGIN
    -- Insert an order
    INSERT INTO dbo.Orders (CustomerID, Amount) VALUES (@CustomerID, 3400.75);
    -- Compare different ways of getting ID
    SELECT
        SCOPE_IDENTITY() AS ScopeIdentity, -- Returns NULL because the insertion was
in the trigger
        @@IDENTITY AS GlobalIdentity,      -- Returns ID from the trigger
        IDENT_CURRENT('dbo.Orders') AS CurrentIdentity; -- Returns the last ID in the
table
    -- Correct way to get OrderID
    DECLARE @OrderID INT = IDENT_CURRENT('dbo.Orders');
    PRINT 'Order number: ' + CAST(@OrderID AS VARCHAR);
END
```

```
-- 5. Error handling
BEGIN TRY
    BEGIN TRANSACTION;
        -- Insert a customer without a required field (will cause an error)
        INSERT INTO dbo.Customers (Email) VALUES ('test@example.com');
        DECLARE @ID INT = SCOPE_IDENTITY();
        PRINT 'This message will not be displayed';
    COMMIT TRANSACTION;
END TRY
BEGIN CATCH
    IF @@TRANCOUNT > 0 ROLLBACK TRANSACTION;
    DECLARE @ErrID INT = SCOPE_IDENTITY();
    PRINT 'Error: ' + ERROR_MESSAGE();
    PRINT 'SCOPE_IDENTITY after error: ' + ISNULL(CAST(@ErrID AS VARCHAR), 'NULL');
END CATCH
```

SERVERPROPERTY

The function returns properties of the current instance

Main capabilities:

- Returns information about server configuration and characteristics
- Allows determining version, edition, and installation parameters
- Useful for diagnostics and administration automation
- Returns NULL for non-existent properties

Input parameters:

- propertyname - name of the requested property

Output parameters:

- Property value
- NULL if the property does not exist

Typical usage scenarios:

- Checking server configuration
- Adapting scripts for different versions
- Logging server characteristics

Query examples:

```
SELECT
    SERVERPROPERTY('MachineName') AS ServerName,
    SERVERPROPERTY('Collation') AS ServerCollation,
    SERVERPROPERTY('Edition') AS Edition,
    SERVERPROPERTY('ProductVersion') AS Version,
    SERVERPROPERTY('ProductLevel') AS ServicePack;
```

TYPE_ID

The function returns the identifier of a system data type by its name.

Main capabilities:

- Returns the ID of a data type in the system
- Works with both system and user types
- Returns NULL if the type does not exist
- Useful for checking type existence

Input parameters:

- type_name - data type name

Output parameters:

- Data type identifier (INT)
- NULL if the type does not exist

Typical usage scenarios:

- Checking existence of a data type
- Getting type ID for other system functions
- Processing user data types

Query examples:

```
-- 1. Getting IDs of standard types
SELECT
    TYPE_ID('int') AS IntTypeID,
    TYPE_ID('varchar') AS VarcharTypeID,
    TYPE_ID('datetime') AS DateTimeTypeID;
```

```
-- 2. Checking type existence before use
DECLARE @TypeName NVARCHAR(128) = 'decimal';
DECLARE @TypeID INT = TYPE_ID(@TypeName);
IF @TypeID IS NOT NULL
    PRINT 'Type ' + @TypeName + ' exists, ID: ' + CAST(@TypeID AS VARCHAR);
ELSE
    PRINT 'Type ' + @TypeName + ' does not exist';
```

```
-- 3. Working with user types
-- Create a user type
IF TYPE_ID('PhoneNumber') IS NULL
    CREATE TYPE dbo.PhoneNumber FROM NVARCHAR(20) NULL;
-- Check its ID
SELECT TYPE_ID('PhoneNumber') AS CustomTypeID;
```

```
-- 4. Getting a list of all types with their IDs
SELECT
    name AS TypeName,
    TYPE_ID(name) AS TypeID,
    system_type_id AS SystemTypeID
FROM sys.types
ORDER BY name;
```

TYPE_NAME

The function returns the name of a data type by its identifier.

Main capabilities:

- Returns the data type name by its ID
- Works with both system and user types
- Returns NULL if a type with the specified ID does not exist
- Useful for converting type ID to a readable name

Input parameters:

- type_id - data type identifier (INT)

Output parameters:

- Data type name (NVARCHAR)
- NULL if the type does not exist

Typical usage scenarios:

- Converting type ID to name
- Analyzing database metadata
- Diagnostics and debugging

Query examples:

```
-- 1. First, get the IDs of needed types
DECLARE @IntTypeID INT = TYPE_ID('int');
DECLARE @VarcharTypeID INT = TYPE_ID('varchar');
DECLARE @DateTimeTypeID INT = TYPE_ID('datetime');
-- Check that the types exist
IF @IntTypeID IS NOT NULL AND @VarcharTypeID IS NOT NULL AND @DateTimeTypeID IS NOT
NULL
BEGIN
    -- Get type names by ID
    SELECT
        TYPE_NAME(@IntTypeID) AS IntTypeName,
        TYPE_NAME(@VarcharTypeID) AS VarcharTypeName,
        TYPE_NAME(@DateTimeTypeID) AS DateTimeTypeName;
END
ELSE
    PRINT 'One or more types do not exist';
```

STRING

ASCII

The function returns the ASCII code of the first character of a string.

Main capabilities:

- Returns the numeric code (0-255) of the first character of a string

- Returns 0 for an empty string
- For Unicode characters, returns the lower byte

Input parameters:

- character_expression - string expression

Output parameters:

- ASCII code (INT)
- NULL if the input parameter is NULL

Typical usage scenarios:

- Checking the format of input data (validation)
- Determining the case of the first character
- Filtering strings by starting character
- Processing special characters
- Converting characters to numeric codes

Query examples:

```
-- 1. Creating and populating a character table
CREATE TABLE CharacterSamples (
    SampleID INT PRIMARY KEY,
    Symbol VARCHAR(10) NOT NULL,
    Description NVARCHAR(50)
);
INSERT INTO CharacterSamples VALUES
(1, 'A', 'Latin capital letter A'),
(2, 'z', 'Latin small letter z'),
(3, '0', 'Digit zero'),
(4, '', 'Empty string'),
(5, ' ', 'Space character'),
(6, 'Ya', 'Cyrillic capital letter Ya'),
(7, '$', 'Dollar sign');
```



```
-- Getting ASCII codes
SELECT
    SampleID,
    Symbol,
    Description,
    ASCII(Symbol) AS ASCII_Code
FROM CharacterSamples;
```

```
-- 2. Creating and populating a user accounts table
CREATE TABLE UserAccounts (
    UserID INT PRIMARY KEY,
    Username VARCHAR(50) NOT NULL,
    Email VARCHAR(100)
);
INSERT INTO UserAccounts VALUES
(1, 'Admin', 'admin@example.com'),
(2, 'user1', 'user1@test.org'),
(3, '1Visitor', 'guest@site.com'),
(4, ' tester', 'test@domain.com'),
(5, 'manager', NULL);
```

```
-- Checking usernames
SELECT
    UserID,
    Username,
    CASE
        WHEN ASCII(Username) BETWEEN 65 AND 90 THEN 'Starts with uppercase'
        WHEN ASCII(Username) BETWEEN 97 AND 122 THEN 'Starts with lowercase'
        WHEN ASCII(Username) BETWEEN 48 AND 57 THEN 'Starts with digit'
        ELSE 'Starts with special char'
    END AS NamingPattern
FROM UserAccounts;
```

```
-- 3. Creating and populating a product catalog table
CREATE TABLE ProductCatalog (
    ProductCode VARCHAR(10) PRIMARY KEY,
    ProductName NVARCHAR(100) NOT NULL,
    Price DECIMAL(10,2)
);
INSERT INTO ProductCatalog VALUES
('A100', 'Laptop', 999.99),
('B200', ' smartphone', 599.99),
('C300', '24" Monitor', 249.99),
('D400', 'Keyboard', 49.99),
('E500', ' Mouse', 29.99);
```

```
-- Finding products with naming issues
SELECT *
FROM ProductCatalog
WHERE ASCII(ProductName) = 32; -- Start with a space
```

CHAR

The function converts an ASCII numeric code to a character.

Main capabilities:

- Converts a number from 0 to 255 to the corresponding ASCII character
- Returns a single character of type CHAR(1)
- For values outside the range, returns NULL

Input parameters:

- integer_expression - integer (ASCII code)

Output parameters:

- ASCII character

Query examples:

```
-- 1. Simple conversions
SELECT
    CHAR(65) AS A,      -- 'A'
    CHAR(97) AS a,      -- 'a'
    CHAR(49) AS one,    -- '1'
    CHAR(32) AS space;  -- ' '
```

```
-- 2. Creating a string with a line break
SELECT 'First line' + CHAR(13) + CHAR(10) + 'Second line' AS MultilineText;
```

CHARINDEX

The function returns the position of the first occurrence of a substring in a string.

Main capabilities:

- Searching for a substring in a string
- Case-sensitivity depends on collation parameters
- Returns the position (starting from 1) or 0 if the substring is not found
- Supports text data types

Input parameters:

- substring - substring to find
- string - string to search in
- start_location (optional) - starting position for search

Output parameters:

- Substring position (INT)
- 0 if the substring is not found
- NULL if any of the arguments is NULL

Typical usage scenarios:

- Searching for keywords in text
- Validating string format
- Parsing strings with delimiters

- Finding position for subsequent substring extraction

Query examples:

```
-- 1. Basic search  
SELECT CHARINDEX('world', 'Hello, world!') AS Position; -- 9
```

```
-- 2. Search with starting position specified  
SELECT CHARINDEX('a', 'banana', 3) AS Position; -- 4
```

```
-- 3. Searching for a non-existent substring  
SELECT CHARINDEX('word', 'Hello world') AS Position; -- 0
```

```
-- 4. Example with NULL  
SELECT CHARINDEX('test', NULL) AS Position; -- NULL
```

CONCAT

The function concatenates two or more strings into one.

Main capabilities:

- Concatenates strings without separators
- Automatically converts non-string types to text
- NULL values are treated as empty strings
- Supports all string data types (CHAR, VARCHAR, NCHAR, NVARCHAR)

Input parameters:

- value1, value2,... valueN - values to concatenate (minimum 2 parameters)

Output parameters:

- Concatenated string (VARCHAR(MAX))

Typical usage scenarios:

- Forming full names from separate components
- Creating complex text messages

- Combining text with numeric values

Query examples:

```
-- 1. Basic string concatenation  
SELECT CONCAT('Hello', 'World') AS Result; -- Result: 'HelloWorld'
```

```
-- 2. Concatenating string and number  
SELECT CONCAT('OrderID: ', 10025) AS OrderInfo; -- Result: 'OrderID: 10025'
```

```
-- 3. Concatenating with NULL value  
SELECT CONCAT('Customer: ', NULL, 'Smith') AS CustomerName; -- Result: 'Customer:  
Smith'
```

FORMAT

The function formats a value according to the specified format.

Main capabilities:

- Formats dates, numbers, and strings
- Supports standard and custom formats
- Takes into account regional settings (culture)
- Returns NVARCHAR

Input parameters:

- value - value to format (date/number)
- format - format string
- culture (optional) - language/region code

Output parameters:

- Formatted string (NVARCHAR)

Query examples:

```
-- 1. Formatting a date  
SELECT FORMAT(GETDATE(), 'dd.MM.yyyy') AS DateFormatted; -- '25.12.2023'
```

```
-- 2. Formatting a number  
SELECT FORMAT(1234567.89, 'N2') AS NumberFormatted; -- '1,234,567.89'
```

```
-- 3. Formatting with culture specified  
SELECT FORMAT(GETDATE(), 'D', 'en-US') AS EnglishDate; -- 'Thursday, February 19,  
2024'
```

LEFT

The function returns the specified number of characters from the beginning of a string.

Main capabilities:

- Extracts part of a string starting from the first character
- Supports all string data types *Input parameters:*
- character_expression - source string
- integer_expression - number of characters to return

Output parameters:

- Part of the string (of the same type as the input string)
- NULL if any of the parameters is NULL

Typical usage scenarios:

- Extracting prefixes (first N characters)
- Truncating long text for display
- Processing strings of fixed format
- Extracting codes and identifiers
- Preparing data for systems with length restrictions

Query examples:

```
-- 1. Basic usage (returns first 4 characters)
SELECT LEFT('Database', 4) AS Result; -- 'Data'
```

```
-- 2. Extracting first 3 characters (less than string length)
SELECT LEFT('SQL', 3) AS Result; -- 'SQL'
```

```
-- 3. Requesting more characters than in the string
SELECT LEFT('Hello', 10) AS Result; -- 'Hello'
```

```
-- 4. Working with NULL values
SELECT LEFT(NULL, 5) AS Result; -- NULL
SELECT LEFT('Text', NULL) AS Result; -- NULL
```

LEN

The function returns the number of characters in a string.

Main capabilities:

- Counts the number of characters in a string
- Ignores trailing spaces (unlike DATALENGTH)
- Returns an integer (INT)

Input parameters:

- string_expression - string to analyze

Output parameters:

- Number of characters (INT)
- NULL if the input parameter is NULL

Typical usage scenarios:

- Checking the length of input data
- Validating strings before processing
- Analyzing text fields in a database

- Optimizing storage of string data

Query examples:

```
-- 1. Basic example  
SELECT LEN('SQL Server') AS Length; -- 10
```

```
-- 2. String with spaces  
SELECT LEN(' Test ') AS Length; -- 6 (trailing spaces ignored)
```

```
-- 3. Empty string  
SELECT LEN('') AS Length; -- 0
```

```
-- 4. With NULL value  
SELECT LEN(NULL) AS Length; -- NULL
```

LOWER

The function converts all characters of a string to lowercase.

Main capabilities:

- Converts all characters of a string to lowercase (small letters)
- Supports all string data types
- Preserves the original data type
- Does not change characters that do not have a lowercase variant

Input parameters:

- `character_expression` - string to convert

Output parameters:

- String in lowercase (of the same type as the input)
- NULL if the input parameter is NULL

Typical usage scenarios:

- Normalizing data before comparison

- Preparing data for case-insensitive search
- Processing user input
- Converting strings to a unified storage format
- Creating case-insensitive indexes

Query examples:

```
-- 1. Basic conversion  
SELECT LOWER('SQL Server') AS LowerCase; -- 'sql server'
```

```
-- 2. Converting string with mixed case  
SELECT LOWER('HeLlO WoRLD') AS LowerCase; -- 'hello world'
```

```
-- 3. Working with NULL value  
SELECT LOWER(NULL) AS LowerCase; -- NULL
```

```
-- 4. Converting national characters  
SELECT LOWER('HELLO') AS LowerCase; -- 'hello'
```

LTRIM

The function removes leading spaces (from the left) from a string.

Main capabilities:

- Removes all space characters at the beginning of a string
- Preserves the original data type
- Does not change spaces inside the string or at the end

Input parameters:

- `character_expression` - string to process

Output parameters:

- String without leading spaces (of the same type)
- NULL if the input parameter is NULL

Typical usage scenarios:

- Cleaning user input
- Preparing data before comparison
- Processing imported data
- Normalizing strings before storage
- Removing unwanted spaces

Query examples:

```
-- 1. Basic usage
SELECT LTRIM('  Hello') AS Result; -- 'Hello'
```

```
-- 2. String with internal spaces
SELECT LTRIM('  SQL  Server  ') AS Result; -- 'SQL  Server  '
```

```
-- 3. Empty string
SELECT LTRIM('') AS Result; -- ''
```

```
-- 4. With NULL value
SELECT LTRIM(NULL) AS Result; -- NULL
```

```
-- 5. With tabs and spaces
SELECT LTRIM('  \t  Text') AS Result; -- '\t  Text' (tab preserved)
```

N

The function indicates that a string literal should be interpreted as a Unicode string.

Main capabilities:

- Converts a string literal to Unicode format
- Allows correct work with national alphabets
- Supports all Unicode standard characters
- Required for compatibility with NVARCHAR types

Input parameters:

unicode_string - string literal in quotes

Output parameters:

- Unicode string

Typical usage scenarios:

- Storing and processing multilingual text
- Working with Cyrillic and other non-Latin characters

Query examples:

```
-- 1. Simple usage with Cyrillic  
SELECT N'Hello, world!' AS RussianText;
```

```
-- 2. Usage with Asian characters  
SELECT N'こんにちは' AS JapaneseText;
```

```
-- 3. Special characters  
SELECT N' Unicode characters ' AS SpecialChars;
```

PATINDEX

The function returns the position of the first occurrence of a pattern in a string.

Main capabilities:

- Pattern search
- Supports all text data types
- Case-sensitivity depends on collation parameters
- Returns position (starting from 1) or 0 if pattern not found

Input parameters:

- '%[pattern]%' - search pattern (must include wildcard characters) expression - string to search in

Output parameters:

- Position of first occurrence
- 0 if pattern not found

Typical usage scenarios:

- Validating string format
- Searching for complex patterns in text
- Analyzing unstructured data
- Checking for forbidden characters

Query examples:

```
-- 1. Searching for a digit in a string  
SELECT PATINDEX('%[0-9]%', 'Order 123') AS Position; -- 7
```

```
-- 2. Searching for a word (case-sensitive)  
SELECT PATINDEX('%server%', 'SQL Server') AS Position; -- 0 (not found)
```

```
-- 3. Search with any characters  
SELECT PATINDEX('%a_e%', 'database') AS Position; -- 2
```

QUOTENAME

The function adds delimiters to a string for safe use as an identifier.

Main capabilities:

- Wraps a string with specified delimiter characters
- Escapes special characters inside the string
- Supports standard quote types: [], ", ""
- Converts a string to a correct identifier

Input parameters:

- string - text to convert (up to 128 characters)

- delimiter (optional) - quote character:
- ']' (default, square brackets)
- '"' (double quotes)
- ' (single quotes)

Output parameters:

- String with escaped special characters
- NULL if the input parameter is NULL

Typical usage scenarios:

- Escaping database object names
- Working with names containing special characters

Query examples:

```
-- 1. Standard wrapping in square brackets
SELECT QUOTENAME('column_name') AS result; -- [column_name]
```

```
-- 2. Processing strings with special characters
SELECT QUOTENAME('table]name') AS result; -- [table]]name]
```

```
-- 3. Using different delimiters
SELECT QUOTENAME('table_name', '"') AS result; -- "table_name"
```

REPLACE

The function replaces all occurrences of a specified substring in a string with a new value.

Main capabilities:

- Replaces all matches of a substring in the source string
- Supports text data types
- Preserves the original data type
- Performs replacement for all found occurrences

Input parameters:

- input_string - source string
- substring_to_replace - substring to replace
- replacement_substring - replacement substring

Output parameters:

- Modified string (of the same type as the input)
- NULL if any of the parameters is NULL

Typical usage scenarios:

- Correcting typos in data
- Removing unwanted characters
- Masking confidential information
- Converting data formats

Query examples:

```
-- 1. Simple text replacement
SELECT REPLACE('Database', 'Data', 'System') AS Result;  -- 'Systembase'
```

```
-- 2. Removing characters
SELECT REPLACE('2023-12-31', '-', '') AS Result;  -- '20231231'
```

```
-- 3. Multiple replacement
SELECT REPLACE(REPLACE('a;b;c', ';', ','), 'b', 'x') AS Result;  -- 'a,x,c'
```

```
-- 4. Working with NULL
SELECT REPLACE('Text', NULL, 'Replacement') AS Result;  -- NULL
```

RIGHT

The function returns the specified number of characters from the end of a string.

Main capabilities:

- Extracts part of a string, starting from the end
- Supports all string data types
- Preserves the original data type

Input parameters:

- `character_expression` - source string
- `integer_expression` - number of characters to return

Output parameters:

- Part of the string (of the same type as the input)
- NULL if any of the parameters is NULL

Typical usage scenarios:

- Extracting suffixes and endings
- Processing strings of fixed format
- Extracting file extensions
- Analyzing codes and identifiers
- Preparing data for systems with restrictions

Query examples:

```
-- 1. Basic usage (last 5 characters)
SELECT RIGHT('SQL Server', 5) AS Result; -- 'erver'
```

```
-- 2. Extracting file extension
SELECT RIGHT('document.pdf', 4) AS FileExtension; -- '.pdf'
```

```
-- 3. Requesting more characters than in the string
SELECT RIGHT('Hello', 10) AS Result; -- 'Hello'
```

```
-- 4. Working with NULL values
SELECT RIGHT(NULL, 3) AS Result; -- NULL
SELECT RIGHT('Text', NULL) AS Result; -- NULL
```

RTRIM

The function removes spaces at the end of a string.

Main capabilities:

- Removes all space characters to the right of the text
- Preserves the original data type
- Does not change spaces at the beginning or middle of the string

Input parameters:

- input_string - string to process

Output parameters:

- String without right spaces
- NULL if the input parameter is NULL

Typical usage scenarios:

- Cleaning data after import
- Preprocessing user input
- Normalizing strings before comparison
- Preparing data for systems with fixed format
- Removing unwanted spaces at the end of fields

Query examples:

```
-- 1. Removing trailing spaces
SELECT RTRIM('Text  ') AS Result; -- 'Text'
```

```
-- 2. Mixed spaces (only trailing are removed)
SELECT RTRIM('  SQL  Server  ') AS Result; -- '  SQL  Server'
```

```
-- 3. Empty string remains empty
SELECT RTRIM('') AS Result; -- ''
```



```
-- 4. NULL input gives NULL output  
SELECT RTRIM(NULL) AS Result; -- NULL
```

SPACE

The function generates a string of the specified number of spaces.

Main capabilities:

- Creates a string consisting only of space characters
- Returns a value of type VARCHAR
- Automatically handles invalid values

Input parameters:

- length - number of spaces (integer)

Output parameters:

- String with the specified number of spaces
- Empty string if length = 0
- NULL, if the input parameter is NULL, or has length < 0

Query examples:

```
-- 1. Simply creating spaces  
SELECT SPACE(5); -- Returns '     '
```

```
-- 2. Usage in concatenation  
SELECT 'Hello' + SPACE(3) + 'World'; -- 'Hello   World'
```

```
-- 3. Edge cases  
SELECT SPACE(0); -- ''  
SELECT SPACE(-1); -- ''  
SELECT SPACE(NULL); -- NULL
```

STR

The function converts a numeric value to a string representation.

Main capabilities:

- Converts numeric values to strings
- Allows specifying total length and number of decimal places
- Supports all major numeric data types
- Automatically rounds numbers during conversion

Input parameters:

- `numeric_expression` - numeric value to convert
- `length` (optional) - total length of result (default 10)
- `decimal` (optional) - number of decimal places (default 0)

Output parameters:

- String representation of the number (VARCHAR)
- NULL if the input parameter is NULL

Typical usage scenarios:

- Formatting numbers for output in reports
- Preparing numeric data for export
- Creating strings with fixed number format
- Generating text identifiers based on numbers
- Preparing data for concatenation with text

Query examples:

```
-- 1. Basic number conversion
SELECT STR(123.456) AS Result; -- '      123'
```

```
-- 2. Conversion with length and precision specified
SELECT STR(123.456, 6, 2) AS Result; -- '123.46'
```

```
-- 3. Converting an integer
SELECT STR(42) AS Result; -- '      42'
```

STRING_AGG

The function aggregates string values from a group into one string with the specified separator.

Main capabilities:

- Aggregates string values through the specified separator
- Supports sorting elements before concatenation (WITHIN GROUP)
- Automatically excludes NULL values

Input parameters:

- expression - column to concatenate
- separator - separator (e.g., ', ')
- WITHIN GROUP (ORDER BY) - optional sorting

Output parameters:

- Concatenated string (NVARCHAR(MAX))
- NULL if there are no rows to concatenate

Query examples:

```
-- Create a test table
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Department VARCHAR(50),
    EmployeeName VARCHAR(100),
    HireDate DATE
);
-- Populate with data
INSERT INTO Employees VALUES
(1, 'IT', 'John Smith', '2020-01-15'),
(2, 'IT', 'Jane Doe', '2019-05-20'),
(3, 'HR', 'Mike Johnson', '2021-03-10'),
(4, 'HR', 'Sarah Williams', '2020-11-01'),
(5, 'Finance', 'David Brown', '2018-07-22');
```

```
-- 1. Simple concatenation of names via comma
SELECT STRING_AGG(EmployeeName, ', ') AS AllEmployees
FROM Employees;
```

```
--2. Sorting
SELECT
    Department,
    STRING_AGG(EmployeeName, ', ') WITHIN GROUP (ORDER BY HireDate) AS TeamMembers
FROM Employees
GROUP BY Department;
```

```
-- Create an orders table
CREATE TABLE Orders (
    OrderID INT,
    ProductList VARCHAR(100)
);
-- Populate with test data
INSERT INTO Orders VALUES
(1, 'Milk, Bread, Eggs'),
(2, 'Coffee, Sugar'),
(3, 'Meat, Fish, Vegetables');
```

```
-- 3. Concatenating product lists via semicolon
SELECT STRING_AGG(ProductList, '; ') AS AllProducts
FROM Orders;
```

STRING_ESCAPE

The function escapes special characters in a string for safe use in JSON.

Main capabilities:

- Converts special characters to escape sequences
- Supports only JSON formatting
- Works exclusively with ASCII characters (English alphabet)
- Automatically handles: ", , /, and control characters (\b, \f, \n, \r, \t)

Input parameters:

- input_string - string to process

- 'json' - required parameter (other formats not supported)

Output parameters:

- String with escaped characters
- NULL if the input parameter is NULL

Typical usage scenarios:

- Preparing system paths for JSON documents
- Escaping technical data and logs
- Generating JSON-compatible strings
- Processing errors and system messages
- Preparing data for API

Query examples:

```
-- 1. Escaping a system path
SELECT STRING_ESCAPE('C:\Program Files\App', 'json') AS EscapedPath;
-- Result: "C:\\Program Files\\App"
```

```
-- 2. Escaping quotes
SELECT STRING_ESCAPE('Text with "quotes"', 'json') AS EscapedText;
-- Result: "Text with \"quotes\""
```

```
-- 3. Processing NULL
SELECT STRING_ESCAPE(NULL, 'json') AS NullResult; -- NULL
```

SUBSTRING

The function extracts part of a string starting from the specified position.

Main capabilities:

- Extracts the specified number of characters from a string

Input parameters:

- input_string - source string

- start_position - starting position (≥ 1)
- length - number of characters to extract

Output parameters:

- Extracted substring
- Empty string if length=0

Typical usage scenarios:

- Extracting codes from strings of fixed format
- Parsing complex identifiers
- Processing text data
- Preparing data for systems with restrictions
- Extracting parts of strings for analysis

Query examples:

```
-- 1. Basic substring extraction  
SELECT SUBSTRING('SQL Server', 5, 3) AS Result; -- 'Ser'
```

```
-- 2. Extracting to the end of the string  
SELECT SUBSTRING('Database', 4, 100) AS Result; -- 'abase'
```

TRANSLATE

The function replaces each character in a string according to the specified replacement map.

Main capabilities:

Replaces characters one-to-one according to the passed sets

Input parameters:

- input_string - string to transform
- characters_to_replace - characters to replace
- replacement_characters - corresponding replacement characters

Output parameters:

- Transformed string (of the same type)
- NULL if any parameter is NULL

Typical usage scenarios:

- Replacing character sets in strings
- Text transliteration
- Converting data formats
- Cleaning and normalizing text
- Replacing special characters

Query examples:

```
-- 1. One-to-one character replacement
SELECT TRANSLATE('1A2B3C', '123', 'XYZ') AS Result; -- 'XAYBZC'
```

```
-- 2. Cyrillic transliteration
SELECT TRANSLATE('Hello', 'Hello', 'Privet') AS Result; -- 'Privet'
```

```
-- 3. Replacing special characters
SELECT TRANSLATE('a*b?c!d', '*?!', '---') AS Result; -- 'a-b-c-d'
```

```
-- 4. Processing NULL
SELECT TRANSLATE('Text', NULL, 'ABCD') AS NullResult; -- NULL
```

TRIM

The function removes spaces from the beginning and end of a string.

Main capabilities:

- Removes all space characters from both ends of a string
- Supports CHAR, VARCHAR, NCHAR, NVARCHAR
- Preserves the original data type

- Does not change spaces inside the string

Input parameters:

- character_expression - string to process

Output parameters:

- String without leading and trailing spaces
- NULL if the input parameter is NULL

Typical usage scenarios:

- Cleaning user input
- Normalizing data before comparison
- Preparing strings for saving to the database
- Processing imported data
- Removing unwanted spaces

Query examples:

```
-- 1. Basic usage
SELECT TRIM('  Hello World  ') AS Result; -- 'Hello World'
```

```
-- 2. With spaces inside the string
SELECT TRIM('  SQL  Server  ') AS Result; -- 'SQL  Server'
```

```
-- 3. Empty string
SELECT TRIM('') AS Result; -- ''
```

```
-- 4. Processing NULL
SELECT TRIM(NULL) AS Result; -- NULL
```

UPPER

The function converts all characters of a string to uppercase.

Main capabilities:

- Converts all characters of a string to uppercase letters
- Does not change characters that do not have an uppercase variant

Input parameters:

- character_expression - string to convert

Output parameters:

- String in uppercase (of the same type)
- NULL if the input parameter is NULL

Typical usage scenarios:

- Normalizing data before comparison
- Preparing data for case-insensitive search
- Processing user input
- Converting strings to a unified storage format
- Creating case-insensitive indexes

Query examples:

```
-- 1. Basic conversion
SELECT UPPER('sql server') AS Result; -- 'SQL SERVER'
```

```
-- 2. Converting Cyrillic
SELECT UPPER('hello') AS Result; -- 'HELLO'
```

```
-- 3. Mixed case
SELECT UPPER('Hello World') AS Result; -- 'HELLO WORLD'
```

```
-- 4. Processing NULL
SELECT UPPER(NULL) AS Result; -- NULL
```

17. Built-in functions of MS SQL Server and Oracle Database dialects in PostgreSQL dialect

Digital Q.DataBase supports all original functions and procedures of the PostgreSQL dialect and certain functions of the Oracle Database dialect (more details in the section "8. Built-in packages, functions and procedures of the Oracle Database dialect")

The DBMS also supports the following MS SQL SERVER dialect functions:

1. len
2. datefromparts
3. nchar
4. date_bucket
5. space
6. char
7. REPLICATE
8. CHARINDEX
9. DATEADD
10. DATEDIFF_BIG
11. datalength
12. DATEDIFF
13. CURRENT_TIMESTAMP
14. CURRENT_DATE

The description of these functions is presented below.

LEN

The LEN() function returns the number of characters in the passed string expression, excluding trailing spaces.

Main capabilities:

- Counting the number of characters in a string, excluding trailing spaces.
- Often used to check the length of string data and normalize data.

Input parameters:

- string_expression
- Evaluated string expression. Can be a constant, variable, or data column of type char, varchar, nchar, nvarchar, binary, varbinary.

Output parameters:

- An integer representing the number of characters in the string excluding trailing spaces.
- Return type:
- bigint if the input parameter is of type varchar(max), nvarchar(max) or varbinary(max).
- int for other data types.

Specifics:

- If there are spaces at the end of the string, they are not counted by the LEN() function.
- For counting the actual amount of occupied data, it is better to use the DATALENGTH() function, which returns the number of bytes.

Typical usage scenarios:

- Data normalization and length checking.
- String conversion and quality control of input information.

Query examples:

```
-- Count the length of a string ignoring the trailing space
SELECT LEN('Hello World ') AS Length;
```

```
-- Use LEN to filter strings by length
SELECT ProductName, LEN(ProductName) AS ProductNameLength
FROM Products
WHERE LEN(ProductName) > 10;
```

DATEFROMPARTS

The DATEFROMPARTS function forms a value of type date based on given components: year, month, and day.

Main capabilities:

- Forms a dated value based on separately specified parts (year, month, and day).
- Used for convenient creation of a date from previously known values.

Input parameters:

- year - Year (integer).
- month - Month (integer from 1 to 12).
- day - Day (integer).

Output parameters:

- Value of type date, formed from the specified components.

Specifics:

- Time components are automatically set to the default value (zero time).
- The function raises an error if components are specified incorrectly (e.g., month out of range 1–12).

Typical usage scenarios:

- Creating a date from previously obtained or calculated year, month, and day values.
- Standardizing date format for further calculations and checks.

Query examples:

```
-- Create a date from parts
SELECT DATEFROMPARTS(2023, 10, 5) AS DateExample;
```

```
-- Combine date parts in a query
SELECT CustomerID, OrderDate, DATEFROMPARTS(YEAR(OrderDate), MONTH(OrderDate),
DAY(OrderDate)) AS ReformattedDate
FROM Orders;
```

NCHAR

The NCHAR function returns the Unicode character corresponding to the given integer character code.

Main capabilities:

- Converts a numeric character code to the Unicode character itself.
- Works with Unicode characters represented in the range from 0 to 1,114,111 (UTF-16).

Input parameters:

- integer_expression - integer expression representing the Unicode character code.

Output parameters:

- One Unicode character corresponding to the entered code.
- Return type:
 - nchar(1) if the collation does not support supplementary Unicode characters.
 - nvarchar(2) if the collation supports supplementary Unicode characters (surrogates).

Specifics:

- If the specified code is outside the valid range, the function returns NULL.
- When working with supplementary Unicode characters, it is recommended to use collations that support the supplementary character flag.

Typical usage scenarios:

- Converting numeric character codes to actual Unicode characters.
- Generating characters for output in interfaces and documents.

Query examples:

```
-- Convert character code 248 (Latin letter ø) to the character  
SELECT NCHAR(248) AS Symbol;
```

SPACE

The SPACE function returns a string of spaces of the specified length.

Main capabilities:

- Generates a string consisting exclusively of space characters.
- Allows easy insertion of the required number of spaces between string elements.

Input parameters:

- integer_expression: A positive integer denoting the required number of spaces.

Output parameters:

- String containing the specified number of spaces.
- Return type: varchar(n), where n equals the value of the input parameter.

Typical usage scenarios:

- Forming aligned text by adding the necessary number of spaces.
- Adding indents before text for better readability.

Query examples:

```
-- Adds two spaces between the employee's last name and first name
SELECT RTRIM(LastName) + SPACE(2) + LTRIM(FirstName)
FROM Person
ORDER BY LastName, FirstName;
```

CHAR

The CHAR function returns a single-byte character with the specified integer code, as defined in the character set and collation of the current database.

Main capabilities:

- Converts a numeric character code to a single-character string.
- Supports work with most common character sets, including ASCII.

Input parameters:

- integer_expression: An integer from 0 to 255 representing the character code.

Output parameters:

- Character corresponding to the specified integer code.
- Return type: char(1).

Typical usage scenarios:

- Inserting control characters (tab, carriage return, new line) into text strings.
- Representing characters by their numeric codes.

Query examples:

```
-- Getting characters by ASCII code  
SELECT CHAR(65)
```

REPLICATE

The REPLICATE function repeats a string value the specified number of times.

Main capabilities:

- Repeats the specified string a certain number of times.
- Can be used to fill missing positions with digits or characters.

Input parameters:

- string_expression: Character string expression or binary data type.
- integer_expression: Number of repetitions (integer).

Output parameters:

- String consisting of the specified expression repeated the given number of times.
- Return type: The same type as the string_expression argument.

Typical usage scenarios:

- Formatting fixed-width strings.
- Generating sequences of identical characters.

Query examples:

```
SELECT REPLICATE('*', 10); -- Will return *****
```

CHARINDEX

The CHARINDEX function searches for one character expression inside another and returns the starting position of the found expression.

Main capabilities:

- Determines the position of a substring in a string.
- Used to check for the presence of a specific sequence of characters.

Input parameters:

- expressionToFind: Substring to find.
- expressionToSearch: Source string to search in.
- start_location: Starting search position (default 1).

Output parameters:

- Position number of the first occurrence of the sought substring.
- Return type: bigint (if the source string is longer than 8000 characters), otherwise int.

Specifics:

- If the expression is not found, returns 0.
- Search is case-sensitive if a case-sensitive collation is specified.
- Collation parameters affect the character matching order.

Query examples:

```
-- Basic substring search
DECLARE @document VARCHAR(64);
SELECT @document = 'Reflectors are vital safety components of your bicycle.';
SELECT CHARINDEX('bicycle', @document);
```

DATEADD

The DATEADD function adds the specified number of time units (datepart) to dates or

timestamps.

Main capabilities:

- Changes a date or time by adding a given interval.
- Uses standard date component notations (year, month, week, etc.).

Input parameters:

- datepart: Date part to which the change is applied (e.g., day, month, hour, etc.).
- number: Number of units to add or subtract (integer).
- date: Date or timestamp to which changes are applied.

Output parameters:

- New date or time value, changed according to the specified interval.
- Return type: Matches the type of the original date argument.

Specifics:

- Rounds fractions of milliseconds, not allowing exceeding the maximum scale.
- Allows the use of different types of arguments (variables, functions, constants).
- Accepts large integers (BIGINT) in some scenarios.

Query examples:

```
-- Basic date increase by one
DECLARE @datetime2 AS DATETIME2 = '2024-01-01 13:10:10.1111111';
SELECT 'year', DATEADD(year, 1, @datetime2)
UNION ALL
SELECT 'month', DATEADD(month, 1, @datetime2)
UNION ALL
SELECT 'day', DATEADD(day, 1, @datetime2);
```

DATEDIFF_BIG

The DATEDIFF_BIG function returns the difference between two dates in the specified time scale (datepart).

Main capabilities:

- Calculates the exact number of time period boundary crossings (year, month, day, etc.) between two dates.
- Returns the result as a large integer (bigint), allowing operations with large time intervals.

Input parameters:

- datepart: Date part by which the difference is calculated (year, quarter, month, day, etc.).
- startdate: Initial date or time value.
- enddate: Final date or time value.

Output parameters:

- Quantitative difference between the start and end dates, measured in the selected time component.
- Return type: bigint.

Specifics:

- Suitable for operations with long intervals (more than 292 years).
- Automatically accounts for transition boundaries between different time units (months, days, hours, etc.).
- Compatible with various date and time representation formats.

Query examples:

```
-- Calculating the difference between two dates  
SELECT DATEDIFF_BIG(day, '2005-12-31 23:59:59', '2006-01-01 00:00:00');
```

DATALENGTH

The DATALENGTH function returns the number of bytes required to store the specified expression.

Main capabilities:

- Measures the physical volume of memory occupied by the given expression.
- Applies to all data types.

Input parameters:

- expression: Any expression whose representation needs to be evaluated.

Output parameters:

- Size of the expression in bytes.
- Return type: bigint for large data types (max) and int for others.

Specifics:

- Differs from the LEN function, which returns the number of characters.
- Takes into account internal data storage formats (Unicode, binary data, etc.).

Query examples:

```
-- Determining the physical length of a column
SELECT length = DATALENGTH(EnglishProductName), EnglishProductName
FROM dbo.DimProduct
ORDER BY EnglishProductName;
```

DATEDIFF

The DATEDIFF function returns the number of elapsed boundaries of a given time interval between two dates.

Main capabilities:

- Calculates the difference between two dates in selected units of measurement (day, month, year, etc.).
- Often used to calculate deadlines and ages of objects.

Input parameters:

- datepart: Interval by which the difference is determined (e.g., day, month, year).
- startdate: First date or timestamp.
- enddate: Second date or timestamp.

Output parameters:

- Difference between two dates in the specified time units.

- Return type: int.

Specifics:

- Uses the beginning and end of the period, excluding the endpoint itself.
- Handles various date formats and time zones.

Query examples:

```
-- Difference between two dates  
SELECT DATEDIFF(day, '2007-05-06', '2007-05-07');
```

CURRENT_TIMESTAMP

The `CURRENT_TIMESTAMP` function returns the current system date and time in datetime format.

Main capabilities:

- Provides current system date and time.
- Convenient for recording the moment of transaction execution or queries.

Input parameters:

No input parameters.

Output parameters:

- Current date and time.
- Return type: datetime.

Specifics:

- Is a non-deterministic function (result changes each time executed).
- Has lower precision than similar functions like `SYSDATETIME`.

Query examples:

```
-- Getting the current date and time  
SELECT CURRENT_TIMESTAMP;
```

CURRENT_DATE

The CURRENT_DATE function returns the current system date in date format.

Main capabilities:

- Designed to get the current date without reference to time zone or time of day.
- Convenient for use in conditions where the date itself is important, not the exact moment in time.

Input parameters:

No input parameters.

Output parameters:

- Only the current date.
- Return type: date.

Specifics:

- Non-deterministic function (each call returns a new date).
- Equivalent to the expression CAST(GETDATE() AS DATE) in terms of ANSI SQL.

Query examples:

```
--Getting the current date  
SELECT CURRENT_DATE;
```

18. Working with the DBMS through external applications

18.1. SQL Server Management Studio

The Digital Q.DataBase DBMS provides support for interaction with SQL Server Management Studio (SSMS). Interaction with SSMS allows users to comfortably work with Digital Q.DataBase using a familiar interface and functionality, which reduces the barrier to learning a new system.

SQL Server Management Studio (SSMS) is a tool from Microsoft designed for managing SQL Server databases. SSMS allows you to administer the database server, create and edit database objects (tables, views, stored procedures), run queries, view and analyze data, configure security, and perform many other administrative tasks.

Main capabilities of SSMS:

- **Creating and managing objects:** creating tables, views, indexes, triggers, stored procedures, and functions.
- **Running queries:** executing SQL queries and viewing results.
- **Import/export data:** transferring data between various sources.
- **Server administration:** configuring security, backing up and restoring data, monitoring performance.
- **Debugging and profiling:** tools for analyzing query performance and identifying bottlenecks.
- **Managing users and roles:** assigning access rights to users and user groups.

SSMS Interface

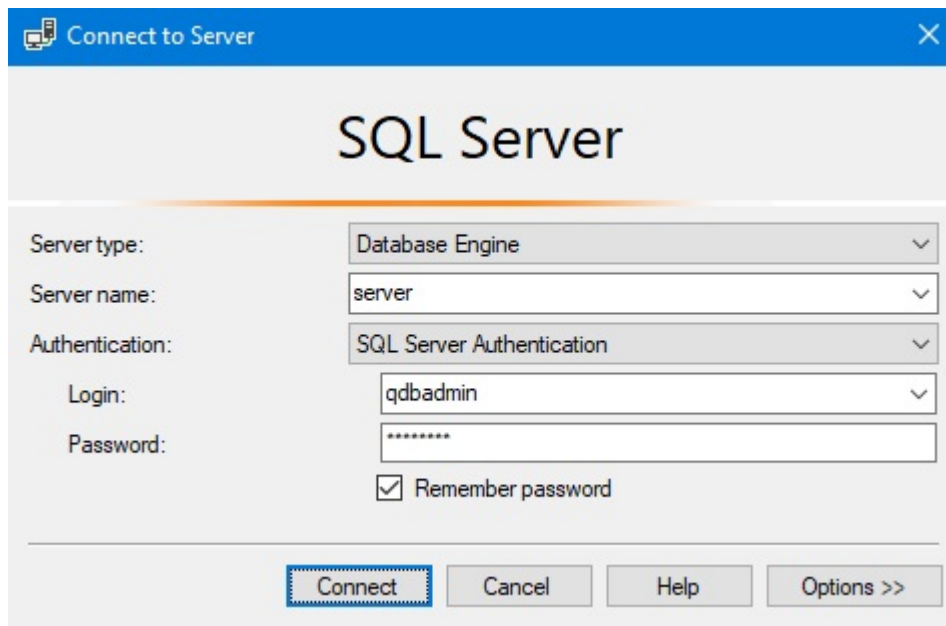


Figure 2. Server connection interface in SSMS

The interface consists of several main parts:

- **Object Explorer:** displaying the structure of database objects (databases, tables, views).
- **Query Editor:** window for writing and executing SQL queries.
- **Window environment:** ability to open multiple windows simultaneously for various tasks.

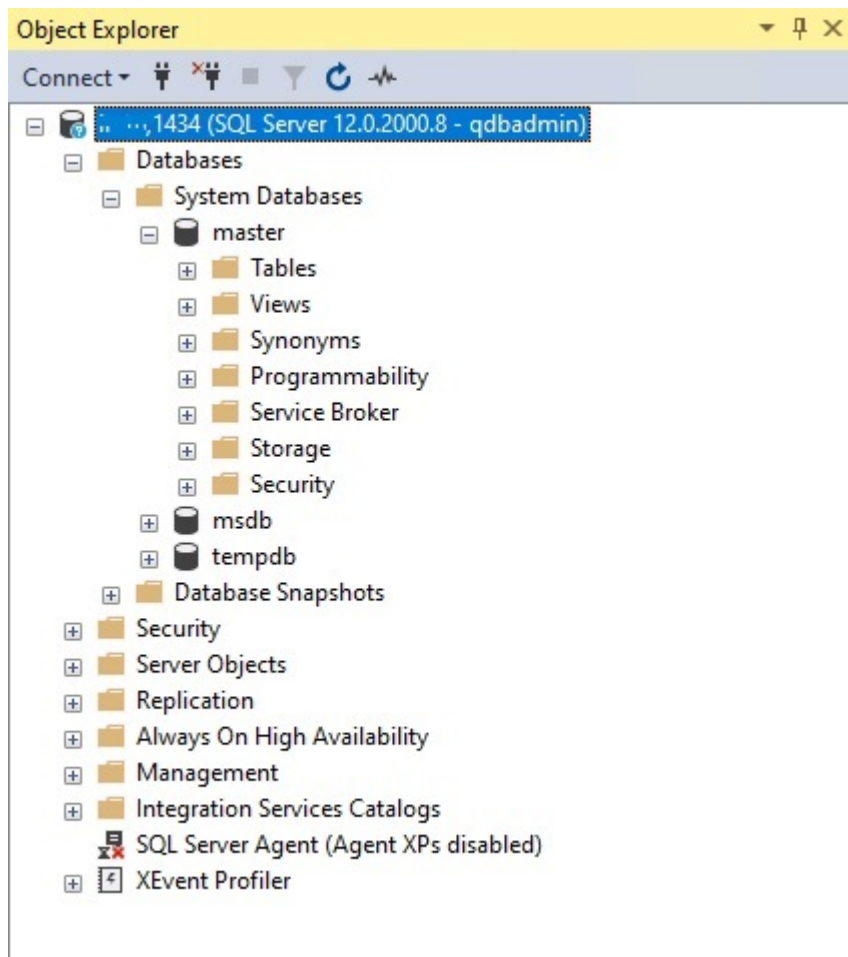
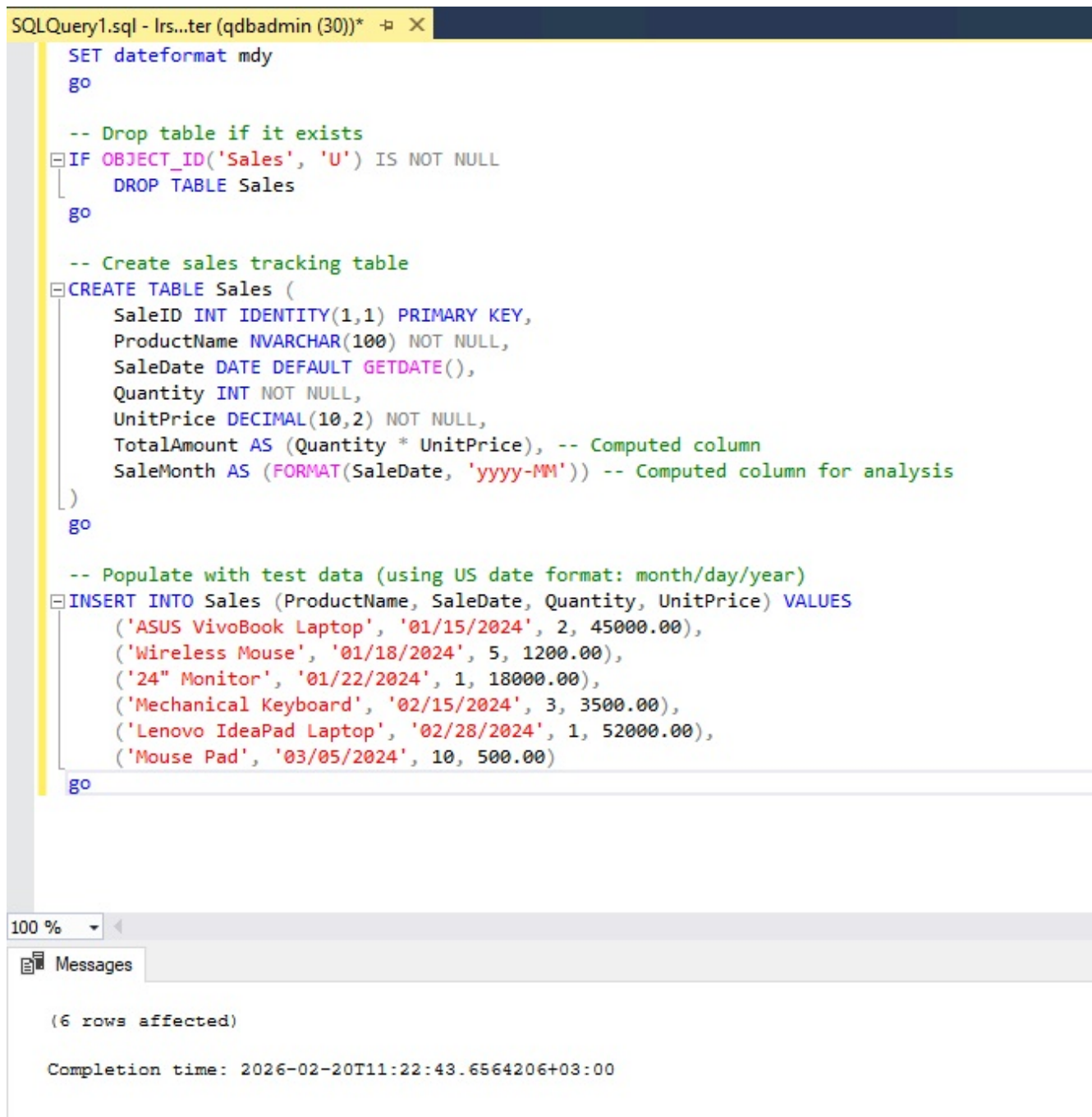


Figure 3. Object Explorer in SSMS



```
SQLQuery1.sql - lrs...ter (qdbadmin (30))* X
SET dateformat mdy
go

-- Drop table if it exists
IF OBJECT_ID('Sales', 'U') IS NOT NULL
    DROP TABLE Sales
go

-- Create sales tracking table
CREATE TABLE Sales (
    SaleID INT IDENTITY(1,1) PRIMARY KEY,
    ProductName NVARCHAR(100) NOT NULL,
    SaleDate DATE DEFAULT GETDATE(),
    Quantity INT NOT NULL,
    UnitPrice DECIMAL(10,2) NOT NULL,
    TotalAmount AS (Quantity * UnitPrice), -- Computed column
    SaleMonth AS (FORMAT(SaleDate, 'yyyy-MM')) -- Computed column for analysis
)
go

-- Populate with test data (using US date format: month/day/year)
INSERT INTO Sales (ProductName, SaleDate, Quantity, UnitPrice) VALUES
    ('ASUS VivoBook Laptop', '01/15/2024', 2, 45000.00),
    ('Wireless Mouse', '01/18/2024', 5, 1200.00),
    ('24" Monitor', '01/22/2024', 1, 18000.00),
    ('Mechanical Keyboard', '02/15/2024', 3, 3500.00),
    ('Lenovo IdeaPad Laptop', '02/28/2024', 1, 52000.00),
    ('Mouse Pad', '03/05/2024', 10, 500.00)
go
```

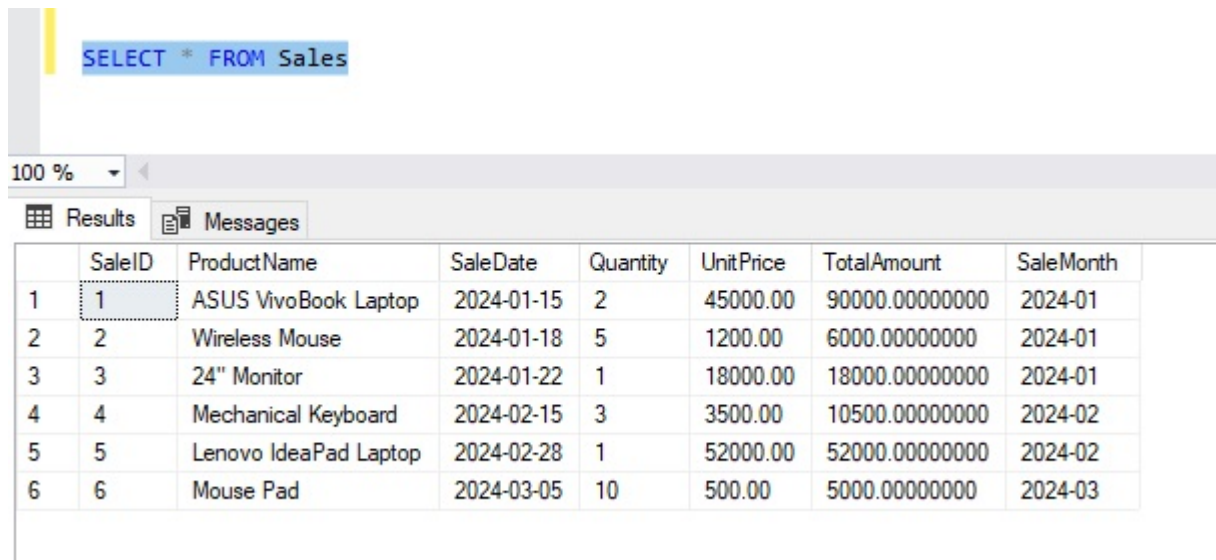
100 %

Messages

(6 rows affected)

Completion time: 2026-02-20T11:22:43.6564206+03:00

Figure 4. SQL query window in SSMS



```
SELECT * FROM Sales
```

	SaleID	ProductName	SaleDate	Quantity	UnitPrice	TotalAmount	SaleMonth
1	1	ASUS VivoBook Laptop	2024-01-15	2	45000.00	90000.00000000	2024-01
2	2	Wireless Mouse	2024-01-18	5	1200.00	6000.00000000	2024-01
3	3	24" Monitor	2024-01-22	1	18000.00	18000.00000000	2024-01
4	4	Mechanical Keyboard	2024-02-15	3	3500.00	10500.00000000	2024-02
5	5	Lenovo IdeaPad Laptop	2024-02-28	1	52000.00	52000.00000000	2024-02
6	6	Mouse Pad	2024-03-05	10	500.00	5000.00000000	2024-03

Figure 5. Demonstration of SQL query work in SSMS

Supported SSMS Versions

Digital Q.DataBase is compatible with the latest versions of SQL Server Management Studio, starting from 2016. Before starting work, make sure your SSMS version meets the minimum requirements and is configured to work with your Digital Q.DataBase instance.

Installing the driver and connecting to Digital Q.DataBase

1. Opening SSMS:

- Launch SQL Server Management Studio on your PC.

2. Connecting to the server:

- In the connection window, select "Object Explorer" and click "Connect".
- In the "Server name" field, specify the address or IP of your Digital Q.DataBase server.
- Select SQL Server Authentication type.
- Enter username and password if necessary.
- Click "Connect".

Working with objects and data

After successful connection, you can start working with objects and data in your database:

1. Viewing objects:

- In the Object Explorer tree, find the node you are interested in (e.g., databases, tables, views).
- Right-click on the desired object and select the required action (open, edit, delete, etc.).

2. Executing queries:

- Open the New Query tab to enter SQL queries for working with data.
- Write a query and press F5 to execute.

19. In-memory component

The in-memory component (DGrid) is an external RAM data storage that operates separately from databases, sessions, and transactions. It is used for temporary storage of values not directly related to operations inside the database. The DGrid package allows users to save (PUT_VALUE) and retrieve (GET_TEXT, GET_BYTEA) data conveniently, independently of the current states of the DBMS. Thanks to scope support, users can control the availability of saved data, making them either globally available or limited to specific users.

Three scopes are available for work:

- GLOBAL — values remain in shared memory and are visible to all users without restrictions;
- QDB_SHARED — values are shared among all QDB users but are allocated in a separate area from global data;
- QDB_USERSCOPE - values are accessible only to the current user, preserved between their sessions, and inaccessible to other users.

A different value in the scope (or its absence) will be automatically replaced with QDB_USERSCOPE.

Set of functions (PostgreSQL dialect) for working with the in-memory component:

dgrid_put_value

The function sets a value for the specified key in the given dgrid space.

Main capabilities:

- Saves text (TEXT) and binary (BYTEA) values
- Supports various storage spaces (scopes)
- Allows organizing structured data storage
- Provides fast access to saved values

Input parameters:

- key - key for saving the value (type TEXT)
- value - value to save (type TEXT or BYTEA)

- scope - optional storage space parameter (type dgrid_scope, default QDB_USERSCOPE)

Output parameters:

- VOID - function does not return a value

Typical usage scenarios:

- Saving user settings
- Caching application data
- Exchanging data between sessions
- Storing binary objects

Query examples:

```
-- Save a key-value pair in the shared QDB_SHARED scope
SELECT dgrid_put_value('text', 'A playful kitty chewed a wire', 'QDB_SHARED');
-- Save a key-value pair in the global GLOBAL scope
SELECT dgrid_put_value('text', 'A lazy corgi stretched for a cutlet', 'GLOBAL');
-- Save a key-value pair in the user scope QDB_USERSCOPE
SELECT dgrid_put_value('text', 'Plush teddy bear 1', 'QDB_USERSCOPE');
-- Absence of scope is equivalent to specifying 'QDB_USERSCOPE'
SELECT dgrid_put_value('text', 'Plush teddy bear 2');
-- Storing binary data
SELECT dgrid_put_value('mykey_bytea', E'\\xCAFE'::bytea);
```

dgrid_get_text

The function allows retrieving a previously set text value by key.

Main capabilities:

- Allows getting text values by key
- Supports working with various storage spaces
- Returns NULL if the value is absent

Input parameters:

- key (TEXT) - key to retrieve the value
- scope (dgrid_scope) - storage space. Default: 'QDB_USERSCOPE'

Output parameters:

- TEXT - found text value
- NULL - if the value is absent

Typical usage scenarios:

- Retrieving saved text data
- Reading user settings
- Accessing shared configurations

Query examples:

```
-- Get a text value by key and scope
SELECT dgrid_get_text('text','QDB_SHARED');
-- Result: 'A playful kitty chewed a wire'
SELECT dgrid_get_text('text','GLOBAL');
-- Result: 'A lazy corgi stretched for a cutlet'
SELECT dgrid_get_text('text');
-- Result: 'Plush teddy bear 2'
SELECT dgrid_get_text('mykey_bytea');
-- Result: \xCAFE
SELECT dgrid_get_text('nonexistent_key');
-- Result: NULL
```

dgrid_get_bytea

The function allows retrieving a previously saved binary value by key from distributed storage.

Main capabilities:

- Retrieves binary data by the specified key
- Supports various storage spaces (scopes)
- Returns NULL if the value is not found
- Preserves the original format of binary data

Input parameters:

- key (TEXT) - key to search for the value
- scope (dgrid_scope) - optional storage space. Default: 'QDB_USERSCOPE'

Output parameters:

- BYTEA - binary data or NULL

Typical usage scenarios:

- Retrieving saved binary objects (images, documents)
- Restoring serialized data
- Accessing shared binary resources
- Working with cached binary data

Query examples:

```
-- Basic example of getting binary data
SELECT dgrid_get_bytea('mykey_bytea');
-- Result: \xcafe
```

```
SELECT dgrid_get_bytea('nonexistent_key');
-- Result: NULL
```

dgrid_has_value

The function checks if a value exists for the given key in the specified scope. Returns TRUE if the value exists, and FALSE otherwise.

Main capabilities:

- Checks for the presence of a value by key in dgrid
- Works with all value types (text and binary)
- Supports various storage spaces (scopes)
- Returns a boolean value (TRUE/FALSE)

Input parameters:

key (TEXT) - key to check for value presence scope (dgrid_scope) - storage space.
Default: QDB_USERSCOPE

Output parameters:

BOOL: * TRUE - if the value exists * FALSE - if the value is absent

Typical usage scenarios:

- Checking data existence before retrieving it
- Validating the presence of mandatory settings
- Checking cached values
- Monitoring the state of distributed storage

Query examples:

```
-- Checking for the presence of an existing value
SELECT dgrid_has_value('text','QDB_SHARED');
-- Result: TRUE
```

```
-- Checking for absence of a value
SELECT dgrid_has_value('nonexistent_key');
-- Result: FALSE
```

dgrid_del_value

The function deletes a value by the specified key from the given storage space.

Main capabilities:

- Deletes values by key from distributed storage
- Supports various storage spaces (scopes)
- Returns the status of the operation
- Does not cause an error if the key is missing

Input parameters:

- key (TEXT) - key to delete
- scope (dgrid_scope) - optional storage space. Default: 'QDB_USERSCOPE'

Output parameters:

BOOLEAN: * TRUE: if the key was successfully deleted or did not exist * FALSE: if the key does not exist in the specified scope

Typical usage scenarios:

- Deleting outdated data
- Cleaning temporary values
- Managing memory of distributed storage
- Synchronizing data between systems

Query examples:

```
--- Basic example of deleting a key
SELECT dgrid_del_value('text','QDB_SHARED');
-- Result: TRUE (successful deletion)
SELECT dgrid_has_value('mykey','QDB_SHARED');
-- Result: FALSE (value absent)
```

```
-- Deleting a non-existent key
SELECT dgrid_del_value('nonexistent_key');
-- Result: FALSE (since the key did not exist)
```

19.1. BCP (fast data loading)

The utility is designed to convert MS SQL database backup files, where the column separator is the '|' character, into binary files obtained from the pg_dump utility with standard data compression.

Arguments for the program can be specified using:

- command line arguments (1st priority);
- config.yaml file (2nd priority);
- for some arguments, a default value is provided.

The list of parameters for the utility is provided by the following list:

Table 1. BCP utility parameters

Parameter	Parameter Argument
--file -f	Name of the file from which data needs to be converted. Default value: file.txt
--resultFile -r	Name of the binary file obtained as a result of conversion by the bcp2dump utility. Default value: result.bin
--table -t	Name of the table into which data needs to be loaded. Default value: newtable_1
--encoding -e	Encoding of the database into which the target binary file will be loaded. Default value: UTF-8
--columns -c	Array of column names in the table specified by the table parameter. Arguments are separated by commas, multi-arguments are supported). Example for config file: columns: ["column1", "column2", "column3", "column4"] Default value: empty array
--encodingDataFrom -a	Encoding from which data should be recoded. Default value: WINDOWS-1251
--encodingDataTo -d	Encoding into which data should be recoded before conversion to binary format. Default value: UTF-8
--debug	Debug support. Supported only in the configuration file. Default value: false.

Example of setting arguments from the command line:

```
./bcp2dump -c column1,column2 -f file.txt -c column3
```

19.2. dblink

The **dblink** module provides connections to other databases from a database session. Below is a description of the module's functions.

Function **dblink_connect**

The **dblink_connect** function establishes a connection to a remote database. The target server and database are specified in a standard libpq connection string. If needed, a name can be assigned to this connection. Multiple named connections can be open at one time, but only one unnamed connection. The connection will persist until closed or until the database session ends. The connection string can also specify the name of an existing foreign server.

Function syntax:

```
dblink_connect(text connstr) returns text  
dblink_connect(text connname, text connstr) returns text
```

Here **connname** is the name assigned to this connection; if omitted, an unnamed connection is opened, replacing any previously existing unnamed connection; **connstr** is a libpq-style connection string, e.g., "hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd options=-csearch_path=". It can also specify a foreign server name.

The function returns a status (this is always the string "OK", as the function raises an exception in case of any error).

If the database, which is not aligned with the safe schema usage pattern, is accessible to untrusted users, start the session by removing writable schemas from the search_path. For example, you can add "options=-csearch_path=" to **connstr**.

Creating connections that do not require password authentication using **dblink_connect** is allowed only for superusers. If ordinary users need this capability, the **dblink_connect_u** function should be used.

Using equals signs in connection names is not recommended, as it can cause confusion with connection strings in other **dblink** functions.

Example 1

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 2

```
SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 3

```
-- Foreign Data Wrapper functionality
-- Note: For this to work, password authentication is required for local connections
-- Otherwise, calling dblink_connect() will result in an error:
-- ERROR: password is required
-- DETAIL: Non-superuser cannot connect if the server does not request a
-- password.
-- HINT: Target server's authentication method must be changed.
--
-- ERROR: password required
-- DETAILS: A regular user cannot connect if the server does not require a password.
-- HINT: The target server's authentication method must be changed.
CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr '127.0.0.1',
dbname 'contrib_regression');
CREATE USER regress_dblink_user WITH PASSWORD 'secret';
CREATE USER MAPPING FOR regress_dblink_user SERVER fdtest OPTIONS (user
'regress_dblink_user', password 'secret');
GRANT USAGE ON FOREIGN SERVER fdtest TO regress_dblink_user;
GRANT SELECT ON TABLE foo TO regress_dblink_user;
\set ORIGINAL_USER :USER
\c - regress_dblink_user
SELECT dblink_connect('myconn', 'fdtest');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 4

```
SELECT * FROM dblink('myconn', 'SELECT * FROM foo') AS t(a int, b text, c text[]);
```

Expected result:

```

a | b | c
---+---+-----
0 | a | {a0,b0,c0}
1 | b | {a1,b1,c1}
2 | c | {a2,b2,c2}
3 | d | {a3,b3,c3}
4 | e | {a4,b4,c4}
5 | f | {a5,b5,c5}
6 | g | {a6,b6,c6}
7 | h | {a7,b7,c7}
8 | i | {a8,b8,c8}
9 | j | {a9,b9,c9}
10 | k | {a10,b10,c10}
(11 rows)

```

Example 5

```

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM regress_dblink_user;
REVOKE SELECT ON TABLE foo FROM regress_dblink_user;
DROP USER MAPPING FOR regress_dblink_user SERVER fdtest;
DROP USER regress_dblink_user;
DROP SERVER fdtest;

```

Function `dblink_connect_u`

The **`dblink_connect_u`** function is no different from **`dblink_connect`**, except that it allows ordinary users to connect with any authentication method.

If the remote server chooses a passwordless authentication mode, impersonation and subsequent privilege escalation are possible because the session will be established as the user running the local Q.DataBase process. Furthermore, even if the remote server requests a password, that password can be obtained from the server environment, for example, from the `~/.pgpass` file belonging to the server user. This can lead not only to impersonation but also to giving the password to an untrusted remote server. Therefore, **`dblink_connect_u`** is initially set up so that the PUBLIC role is denied all rights to use it, meaning only superusers can call it. In some situations, it may be acceptable to grant EXECUTE rights for **`dblink_connect_u`** to specific trusted users, but this must be done carefully. It is also recommended to ensure that the `~/.pgpass` file belonging to the server user does *not* contain any entries with an asterisk as the host name.

Function syntax:

```
dblink_connect_u(text connstr) returns text  
dblink_connect_u(text connname, text connstr) returns text
```

Function **dblink_disconnect**

dblink_disconnect closes a connection previously opened by the **dblink_connect** function. The form without arguments closes the unnamed connection.

Function syntax:

```
dblink_disconnect() returns text  
dblink_disconnect(text connname) returns text
```

Here **connname** is the name of the named connection to close.

The function returns a status (this is always the string "OK", as the function raises an exception in case of any error).

Example 1

```
SELECT dblink_disconnect();
```

Expected result:

```
dblink_disconnect  
-----  
OK  
(1 row)
```

Example 2

```
SELECT dblink_disconnect('myconn');
```

Expected result:

```
dblink_disconnect
-----
OK
(1 row)
```

Function dblink

The **dblink** function executes a query (usually **SELECT**, but it can be any other SQL statement returning rows) in a remote database.

When two arguments of type text are passed to this function, the first is first considered as the name of a persistent connection; if such a connection is found, the command is executed on it. If not found, the first argument is treated as a connection string, as for the **dblink_connect** function, and the specified connection is established only for the duration of this command.

Function syntax:

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
```

```
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
```

```
dblink(text sql [, bool fail_on_error]) returns setof record
```

Here **connname** is the name of the connection to use; omit this parameter to use the unnamed connection; **connstr** is the connection string described earlier for **dblink_connect**; **sql** is the SQL query to execute in the remote database, e.g., "SELECT * FROM foo"; **fail_on_error** - if **true** (default), then if an error occurs on the remote side of the connection, the error is also raised locally. If **false**, the remote error is reported locally as a NOTICE, and the function returns no rows.

This function returns the rows produced by the query. Because **dblink** can execute arbitrary queries, it is declared as returning type **record**, not some specific set of columns. This means that you must specify the expected set of columns in the calling query - otherwise Q.DataBase will not know what to expect.

Example 1


```
SELECT *
FROM dblink('dbname=mydb options=-csearch_path=',
'SELECT proname, prosrc FROM pg_proc')
AS t1(proname name, prosrc text)
WHERE proname LIKE 'bytea%';
```

In the "alias" part of the **FROM** clause, column names and types that the function will return must be specified. This allows the system to understand what ***** should expand to, and what **proname** refers to in the **WHERE** clause, before trying to execute the function. At runtime, an error will occur if the actual query result from the remote database does not contain as many columns as specified in the **FROM** clause. However, column names and types may not match. The function will succeed if the returned data rows are valid input for the column type declared in the **FROM** clause.

Using **dblink** with predefined queries will be more convenient if you create a view. This will hide column type information in its definition and avoid writing it out in every query.

Example 1

```
CREATE VIEW myremote_pg_proc AS
SELECT *
FROM dblink('dbname=postgres options=-csearch_path=',
'SELECT proname, prosrc FROM pg_proc')
AS t1(proname name, prosrc text);
SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
Examples
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
'SELECT proname, prosrc FROM pg_proc')
AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
```

Expected result:

```
proname | prosrc
-----+-----
byteacat | byteacat
byteaeq  | byteaeq
bytealt  | bytealt
byteale  | byteale
byteagt  | byteagt
byteage  | byteage
byteane  | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteain  | byteain
byteaout | byteaout
(12 rows)
```

Example 2

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 3

```
SELECT * FROM dblink('SELECT proname, prosrc FROM pg_proc')
AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
```

Expected result:

```
proname | prosrc
-----+-----
byteacat | byteacat
byteaeq  | byteaeq
bytealt  | bytealt
byteale  | byteale
byteagt  | byteagt
byteage  | byteage
byteane  | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteain  | byteain
byteaout | byteaout
(12 rows)
```

Example 4

```
SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 5

```
SELECT * FROM dblink('myconn', 'SELECT proname, prosrc FROM pg_proc')
AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
```

Expected result:

```

praname | prosrc
-----+-----
bytearecv | bytearecv
byteasend | byteasend
byteale | byteale
byteagt | byteagt
byteage | byteage
byteane | byteane
byteacmp | byteacmp
bytealike | bytealike
byteanlike | byteanlike
byteacat | byteacat
byteaeq | byteaeq
bytealt | bytealt
byteain | byteain
byteaout | byteaout
(14 rows)

```

Function **dblink_exec**

The **dblink_exec** function executes a command (i.e., any SQL statement not returning rows) in a remote database.

When two arguments of type **text** are passed to this function, the first is first considered as the name of a persistent connection; if such a connection is found, the command is executed on it. If not found, the first argument is treated as a connection string, as for the **dblink_connect** function, and the specified connection is established only for the duration of this command.

Function syntax:

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
```

```
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
```

```
dblink_exec(text sql [, bool fail_on_error]) returns text
```

Here **connname** is the name of the connection to use; omit this parameter to use the unnamed connection; **connstr** is the connection string described earlier for **dblink_connect**; **sql** is the SQL query you want to execute in the remote database, e.g.,

"INSERT INTO FOO VALUES(0, 'a', '{"a0","b0","c0"}')"; **fail_on_error** - if **TRUE** (default), then if an error occurs on the remote side of the connection, the error is also raised locally. If **FALSE**, the remote error is reported locally as a NOTICE, and the function's return value will be ERROR.

The function returns a status (either the command status string, or ERROR).

Example 1

```
SELECT dblink_connect('dbname=dblink_test_standby');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 2

```
SELECT dblink_exec('INSERT INTO foo VALUES(21, ''z'', '{"a0","b0","c0"}');');
```

Expected result:

```
dblink_exec
-----
INSERT 943366 1
(1 row)
```

Example 3

```
SELECT dblink_connect('myconn', 'dbname=regression');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 4

```
SELECT dblink_exec('myconn',
'INSERT INTO foo VALUES (21, 'z', '{"a0","b0","c0"}');');
```

Expected result:

```
dblink_exec
-----
INSERT 6432584 1
(1 row)
```

Example 5

```
SELECT dblink_exec('myconn', 'INSERT INTO pg_class VALUES ('foo'),false);
```

Expected result:

```
NOTICE: sql error
DETAIL: ERROR: null value in column "relnamespace" violates not-null constraint
dblink_exec
-----
ERROR
(1 row)
```

Function dblink_open

The **dblink_open** function opens a cursor in a remote database. The opened cursor can be manipulated by the **dblink_fetch** and **dblink_close** functions.

Function syntax:

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns
text
```

Here **connname** is the name of the connection to use; omit this parameter to use the unnamed connection; **cursorname** is the name assigned to the cursor; **sql** is the **SELECT** statement you want to execute in the remote database, e.g., "SELECT * FROM pg_class"; **fail_on_error** - if **TRUE** (default), then if an error occurs on the remote side of the connection, the error is also raised locally. If **FALSE**, the remote error is reported locally as a NOTICE, and the function's return value will be ERROR.

The function returns a status, OK or ERROR.

Since a cursor can only exist within a transaction, the **dblink_open** function starts an explicit transaction block (with a **BEGIN** command) on the remote side if a transaction is not already open there. This transaction will be closed again upon the corresponding call to **dblink_close**. Note that if you modify data with **dblink_exec** between calls to **dblink_open** and **dblink_close**, and then an error occurs, or if you call **dblink_disconnect** before **dblink_close**, your changes *will be lost* because the transaction will be aborted.

Example 1

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 2

```
SELECT dblink_open('foo', 'SELECT proname, prosrc FROM pg_proc');
```

Expected result:

```
dblink_open
-----
OK
(1 row)
```

Function **dblink_fetch**

The **dblink_fetch** function fetches rows from a cursor previously opened by the **dblink_open** function.

Function syntax:

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error])
returns setof record
```

Here **connname** is the name of the connection to use; omit this parameter to use the unnamed connection; **cursorname** is the name assigned to the cursor; **howmany** is the maximum number of rows to fetch. This function fetches the next **howmany** rows through the cursor, starting at the current cursor position and moving forward. When the cursor reaches the end, no more rows are produced; **fail_on_error** - if **TRUE** (default), then if an error occurs on the remote side of the connection, the error is also raised locally. If **FALSE**, the remote error is reported locally as a NOTICE, and the function returns no rows.

This function returns the rows fetched through the cursor. To use this function, you must specify the expected set of columns, as mentioned earlier in the description of **dblink**.

If the number of returned columns defined in the **FROM** clause does not match the actual number of columns returned by the remote cursor, an error is raised. In this case, the remote cursor still advances by as many rows as it would have if the error had not occurred. The same is true for any other errors occurring during local processing of results after executing the remote **FETCH** command.

Example 1

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
```

Expected result:


```
dblink_connect
-----
OK
(1 row)
```

Example 2

```
SELECT dblink_open('foo', 'SELECT proname, prosrc FROM pg_proc WHERE proname LIKE
''bytea%'');
```

Expected result:

```
dblink_open
-----
OK
(1 row)
```

Example 3

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

Expected result:

```
funcname | source
-----+-----
byteacat | byteacat
byteacmp | byteacmp
byteaeq  | byteaeq
byteage  | byteage
byteagt  | byteagt
(5 rows)
```

Example 4

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

Expected result:

```
funcname | source
-----+-----
byteain  | byteain
byteale  | byteale
bytealike| bytealike
bytealt  | bytealt
byteane  | byteane
(5 rows)
```

Example 5

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

Expected result:

```
funcname | source
-----+-----
byteanlike | byteanlike
byteaout   | byteaout
(2 rows)
```

Example 6

```
SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
```

Expected result:

```
funcname | source
-----+-----
(0 rows)
```

Function dblink_close

The **dblink_close** function closes a cursor previously opened by the **dblink_open** function.

Function syntax:

```
dblink_close(text cursorname [, bool fail_on_error]) returns text  
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

Here **connname** is the name of the connection to use; omit this parameter to use the unnamed connection; **cursorname** is the name of the cursor to close; **fail_on_error** - if **TRUE** (default), then if an error occurs on the remote side of the connection, the error is also raised locally. If **FALSE**, the remote error is reported locally as a NOTICE, and the function's return value will be ERROR.

The function returns a status, OK or ERROR.

If the **dblink_open** call started an explicit transaction block and this is the last open cursor remaining in this connection, then **dblink_close** will execute the corresponding **COMMIT** command.

Example 1

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
```

Expected result:

```
dblink_connect  
-----  
OK  
(1 row)
```

Example 2

```
SELECT dblink_open('foo', 'SELECT proname, prosrc FROM pg_proc');
```

Expected result:

```
dblink_open  
-----  
OK  
(1 row)
```

Example 3

```
SELECT dblink_close('foo');
```

Expected result:

```
dblink_close
-----
OK
(1 row)
```

Function **dblink_get_connections**

The **dblink_get_connections** function returns an array of names of all open named **dblink** connections.

Function syntax:

```
dblink_get_connections() returns text
```

The function returns a text array of connection names, or **NULL** if they are absent.

Example 1

```
SELECT dblink_get_connections();
```

Function **dblink_error_message**

The **dblink_error_message** function retrieves the most recent remote error message for the given connection.

Function syntax:

```
dblink_error_message(text connname) returns text
```

Here **connname** is the name of the connection to use.

The function returns the last error message or OK if there have been no errors in this connection's session.

When asynchronous queries are sent by the **dblink_send_query** function, the error

message associated with the current connection may not change until a response message is received from the server. Therefore, you should usually call the **dblink_is_busy** or **dblink_get_result** function before calling **dblink_error_message** so that an error that occurred during execution of an asynchronous query is not missed.

Example 1

```
SELECT dblink_error_message('dtest1');
```

Function **dblink_send_query**

The **dblink_send_query** function sends a query for asynchronous execution, i.e., does not wait for the result. There should be no already executing asynchronous query associated with this connection.

After successfully sending an asynchronous query, its completion status can be checked by calling the **dblink_is_busy** function, and eventually the data can be obtained by calling **dblink_get_result**. You can also try to cancel an active asynchronous query by calling **dblink_cancel_query**.

Function syntax:

```
dblink_send_query(text connname, text sql) returns int
```

Here **connname** is the name of the connection to use; **sql** is the SQL statement to execute in the remote database, e.g., "SELECT * FROM pg_class".

The function returns 1 if the query was successfully sent for processing, or 0 otherwise.

Example 1

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

Function **dblink_is_busy**

The **dblink_is_busy** function checks whether an asynchronous query is being executed through the connection.

Function syntax:

```
dblink_is_busy(text connname) returns int
```

Here **connname** is the name of the connection to check.

The function returns 1 if the connection is busy, or 0 otherwise. If this function returns 0, it is guaranteed that a call to **dblink_get_result** will not block.

Example 1

```
SELECT dblink_is_busy('dtest1');
```

Function **dblink_get_notify**

The **dblink_get_notify** function returns notifications either from the unnamed connection or from the connection with the given name. To receive notifications via **dblink**, you must first execute **LISTEN** using the **dblink_exec** function.

Function syntax:

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)  
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra  
text)
```

Here **connname** is the name of the named connection whose notifications should be received.

The function returns **setof (notify_name text, be_pid int, extra text)** or an empty set if there are no notifications.

Example 1

```
SELECT dblink_exec('LISTEN virtual');
```

Expected result:

```
dblink_exec
-----
LISTEN
(1 row)
```

Example 2

```
SELECT * FROM dblink_get_notify();
```

Expected result:

```
notify_name | be_pid | extra
-----+-----+-----
(0 rows)
```

Example 3

```
NOTIFY virtual;
NOTIFY
SELECT * FROM dblink_get_notify();
```

Expected result:

```
notify_name | be_pid | extra
-----+-----+-----
virtual | 1229 |
(1 row)
```

Function dblink_get_result

The **dblink_get_result** function receives the results of an asynchronous query previously launched by a call to **dblink_send_query**. If the query is still executing, **dblink_get_result** will wait for it to complete.

Function syntax:

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

Here **connname** is the name of the connection to use; **fail_on_error** - if **TRUE** (default), then if an error occurs on the remote side of the connection, the error is also raised locally. If **FALSE**, the remote error is reported locally as a NOTICE, and the function returns no rows.

For an asynchronous query (i.e., an SQL statement returning rows), this function outputs the rows obtained as a result of the query. To use this function, you must specify the expected set of columns, as mentioned earlier in the description of **dblink**.

For an asynchronous command (i.e., an SQL statement not returning rows), this function returns one row with one text column containing the command status string. For such a call, you must also define in the **FROM** clause that the result will contain one text column.

This function *must* be called if **dblink_send_query** returns 1. It must be called once for each sent query, and then once more to get an empty result set, before the connection can be used again.

When using **dblink_send_query** and **dblink_get_result**, the **dblink** subsystem receives the entire remote result set before passing it for local processing. If the query returns a large number of rows, this can occupy a lot of memory in the local session.

Therefore, it may be better to open such a query as a cursor by calling **dblink_open**, and then fetch results in manageable chunks. Alternatively, you can use the simple **dblink** function, which does not allow memory filling by spilling large result sets to disk.

Example 1

```
SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
```

Expected result:

```
dblink_connect
-----
OK
(1 row)
```

Example 2


```
SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3') AS
t1;
```

Expected result:

```
t1
----
1
(1 row)
```

Example 3

```
SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
```

Expected result:

```
f1 | f2 | f3
----+-----+-----
0 | a | {a0,b0,c0}
1 | b | {a1,b1,c1}
2 | c | {a2,b2,c2}
(3 rows)
```

Example 4

```
SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
```

Expected result:

```
f1 | f2 | f3
----+-----+-----
(0 rows)
```

Example 5

```
SELECT * FROM
dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3;
SELECT * FROM foo WHERE f1 > 6') AS t1;
```

Expected result:

```
t1
----
1
(1 row)
```

Example 6

```
SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
```

Expected result:

```
f1 | f2 | f3
----+-----+-----
0 | a | {a0,b0,c0}
1 | b | {a1,b1,c1}
2 | c | {a2,b2,c2}
(3 rows)
```

Example 7

```
SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
```

Expected result:

```
f1 | f2 | f3
----+-----+-----
7 | h | {a7,b7,c7}
8 | i | {a8,b8,c8}
9 | j | {a9,b9,c9}
10 | k | {a10,b10,c10}
(4 rows)
```

Example 8

```
SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text,
f3 text[]);
```

Expected result:

```
f1 | f2 | f3
----+-----+----
(0 rows)
```

Function **dblink_cancel_query**

The **dblink_cancel_query** function attempts to cancel any query executing through the given connection. A call to **dblink_cancel_query** will not necessarily be successful (e.g., because the remote query has already completed). However, the cancellation request increases the likelihood of interrupting the executing query. In this case, you need to complete the normal query processing procedure, for example, by calling **dblink_get_result**.

Function syntax:

```
dblink_cancel_query(text connname) returns text
```

Here **connname** is the name of the connection to use.

The function returns OK if the cancellation request was sent, or an error message text in case of failure.

Example 1

```
SELECT dblink_cancel_query('dtest1');
```

Function **dblink_get_pkey**

The **dblink_get_pkey** function outputs information about the primary key of a relation in the local database. This is sometimes useful when forming queries to be sent to remote databases.

Function syntax:

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

Here **relname** is the name of a local relation, e.g., *foo* or *myschema.mytab*. Enclose it in double quotes if it is a mixed-case name or contains special characters, e.g., "FooBar"; without quotes, this string is converted to lower case.

The function returns one row for each primary key field, or no rows if the relation has no primary key. The resulting row type is defined as

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

The **position** column contains a number from 1 to *N*; this is the field number in the primary key, not the column number in the table's column list.

Example 1

```
CREATE TABLE foobar (
  f1 int,
  f2 int,
  f3 int,
  PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE
SELECT * FROM dblink_get_pkey('foobar');
```

Expected result:

```
position | colname
-----+-----
1 | f1
2 | f2
3 | f3
(3 rows)
```

Function dblink_build_sql_insert

The **dblink_build_sql_insert** function can be useful for selective replication of a local table with a remote database. It selects a row from the local table by the given primary

key, and then forms an SQL **INSERT** command duplicating that row, but replaces the primary key values with data from the last argument. To get an exact copy of the row, simply specify the same values in the last two arguments.

Function syntax:

```
dblink_build_sql_insert(text relname,  
int2vector primary_key_attnums,  
integer num_primary_key_atts,  
text[] src_pk_att_vals_array,  
text[] tgt_pk_att_vals_array) returns text
```

Here **relname** is the name of a local relation, e.g., *foo* or *myschema.mytab*. Enclose it in double quotes if it is a mixed-case name or contains special characters, e.g., "FooBar"; without quotes, this string is converted to lower case; **primary_key_attnums** are the numbers (starting from 1) of the primary key field attributes, e.g., 1 2; **num_primary_key_atts** is the number of primary key fields; **src_pk_att_vals_array** are the primary key field values by which the local tuple will be searched. Each field is represented here in text form. If there is no local row with these primary key values, an error is raised; **tgt_pk_att_vals_array** are the primary key field values that will be placed in the resulting **INSERT** command. Each field is represented in text form.

The function returns the requested SQL statement as text.

Attribute numbers in **primary_key_attnums** are treated as logical column numbers corresponding to column positions in **SELECT * FROM relname**.

Example 1

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b''a"}');
```

Expected result:

```
dblink_build_sql_insert  
-----  
INSERT INTO foo(f1,f2,f3) VALUES('1','b''a','1')  
(1 row)
```

Function dblink_build_sql_delete

The **dblink_build_sql_delete** function can be useful for selective replication of a local table with a remote database. It builds a **DELETE** SQL command that will delete the row with the specified primary key values.

Function syntax:

```
dblink_build_sql_delete(text relname,  
int2vector primary_key_attnums,  
integer num_primary_key_atts,  
text[] tgt_pk_att_vals_array) returns text
```

Here **relname** is the name of a local relation, such as *foo* or *myschema.mytab*. Enclose it in double quotes if it is a mixed-case name or contains special characters, e.g., "FooBar"; without quotes this string is folded to lower case; **primary_key_attnums** are the attribute numbers (starting from 1) of the primary key fields, e.g., 1 2; **num_primary_key_atts** is the number of primary key fields; **tgt_pk_att_vals_array** are the values of the primary key fields to be used in the resulting **DELETE** command. Each field is represented in text form.

The function returns the requested SQL statement as text.

The attribute numbers in **primary_key_attnums** are treated as logical column numbers corresponding to the column positions in **SELECT * FROM relname**. Previous versions treated these numbers as physical column positions. The difference between these approaches becomes apparent when columns to the left of the specified ones are dropped during the table's lifetime.

Example 1

```
SELECT dblink_build_sql_delete('MyFoo', '1 2', 2, '{"1", "b"}');
```

Expected result:

```
dblink_build_sql_delete  
-----  
DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'  
(1 row)
```

Function **dblink_build_sql_update**

The **dblink_build_sql_update** function can be useful for selective replication of a local table with a remote database. It selects a row from the local table based on the given primary key, and then builds an **UPDATE** SQL command duplicating that row, but replacing the primary key values with data from the last argument. To get an exact copy of the row, simply specify the same values in the last two arguments. The **UPDATE** command always assigns values to all fields of the row — the main difference between this function and **dblink_build_sql_insert** is that it assumes the target row already exists in the remote table.

Function syntax:

```
dblink_build_sql_update(text relname,  
int2vector primary_key_attnums,  
integer num_primary_key_atts,  
text[] src_pk_att_vals_array,  
text[] tgt_pk_att_vals_array) returns text
```

Here **relname** is the name of a local relation, such as *foo* or *myschema.mytab*. Enclose it in double quotes if it is a mixed-case name or contains special characters, e.g., "FooBar"; without quotes this string is folded to lower case; **primary_key_attnums** are the attribute numbers (starting from 1) of the primary key fields, e.g., 1 2; **num_primary_key_atts** is the number of primary key fields; **src_pk_att_vals_array** are the values of the primary key fields used to search for the local tuple. Each field here is represented in text form. If there is no local row with these primary key values, an error is raised; **tgt_pk_att_vals_array** are the values of the primary key fields to be placed into the resulting **UPDATE** command. Each field is represented in text form.

The function returns the requested SQL statement as text.

The attribute numbers in **primary_key_attnums** are treated as logical column numbers corresponding to the column positions in **SELECT * FROM relname**. Previous versions treated these numbers as physical column positions. The difference between these approaches becomes apparent when columns to the left of the specified ones are dropped during the table's lifetime.

Example 1

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
```

Expected result:

```
dblink_build_sql_update
```

```
-----  
UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'  
(1 row)
```


20. Administration and Security

20.1. User and Access Rights Management

User and access rights management is a critically important part of working with any Database Management System (DBMS). It allows limiting the circle of individuals who have access to data and ensures data safety and confidentiality. The Digital Q.DataBase DBMS provides extensive capabilities for secure user management and ensuring the appropriate level of authorization.

Basic Concepts * **User:** An individual account created for accessing the database. * **Roles:** A set of privileges that can be assigned to users. * **Privileges:** Permitted actions on database objects (tables, views, procedures, etc.) * **Security Policy:** A set of measures and settings aimed at protecting data from unauthorized access.

Stage 1: Creating Users

Creating a new user

```
CREATE USER username IDENTIFIED BY password;
```

For example:

```
CREATE USER admin IDENTIFIED BY secure_password;
```

When an object is created in a database, an owner is assigned to it. The owner is usually the role under which the creation statement was executed. Unlike a user, the database owner (or superuser) has access to all its objects and can perform operations on them. Other users need to be granted rights for access. The set of rights applicable to a particular object depends on the object type (table, function, etc.). The right to modify or delete an object is an inherent right of the object owner; it cannot be revoked or transferred to another.

A new owner can be assigned to an object using the `ALTER` command for the corresponding object type, for example:

```
ALTER TABLE table_name OWNER TO new_owner;
```

A superuser can do this without restrictions, while an ordinary user — only if they are

simultaneously the current owner of the object (or a member of the owner's group role) and a member of the new role. Assigning a role to a user

```
GRANT role_name TO username;
```

For example:

```
GRANT DBA TO admin;
```

To grant rights, the GRANT command is used. The right to update a table can be granted to this role as follows:

```
GRANT UPDATE ON new_table TO username;
```

If ALL is written instead of a specific right, the role will receive all rights applicable to an object of that type.

Stage 2: Assigning Privileges

General database access

```
GRANT ALL PRIVILEGES TO username;
```

For example:

```
GRANT ALL PRIVILEGES TO admin;
```

Access to specific objects

```
GRANT SELECT, INSERT, UPDATE, DELETE ON object_name TO username;
```

For example:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON employees TO hr_manager;
```

To revoke previously granted rights from users, use the REVOKE command:

```
REVOKE ALL ON new_table FROM PUBLIC;
```

Stage 3: Restricting Access

Revoking privileges

```
REVOKE privilege_name ON object_name FROM username;
```

For example:

```
REVOKE DELETE ON employees FROM hr_manager;
```

Full revocation of all privileges

```
REVOKE ALL PRIVILEGES FROM username;
```

For example:

```
REVOKE ALL PRIVILEGES FROM guest_user;
```

Usually, only the object owner (or a superuser) can manage rights. However, it is possible to grant access to an object 'WITH GRANT OPTION', which will allow the recipient to grant it to others. If this grant option is later revoked, all who received this access right (directly or through a chain of grants) will lose it.

SELECT

Allows executing SELECT for any column or listed columns in a given table, view, materialized view, or other table-like object. Also allows executing COPY TO. Additionally, this right is required to reference existing column values in UPDATE or DELETE. For sequences, this right allows using the currval function. For large objects, it allows reading the object's contents.

INSERT

Allows inserting rows via INSERT into a given table, view, etc. Can be granted for specific columns; in this case, only those columns can be assigned values in the INSERT command (other columns will receive default values). Also allows executing COPY

FROM.

UPDATE

Allows modifying data via UPDATE in all, or only listed, columns of a given table, view, etc. (In practice, any non-trivial UPDATE command will also require the SELECT right, as it must access the table's columns to determine which rows are to be modified, and/or to compute new column values. For SELECT ... FOR UPDATE and SELECT ... FOR SHARE, this right is also required for at least one column, in addition to the SELECT right. For sequences, this right allows using the nextval and setval functions. For large objects, this right allows writing data to the object or truncating it.

DELETE

Allows deleting rows via the DELETE command from a table, view, etc. (In practice, any non-trivial DELETE command will also require the SELECT right, as it must access the table's columns to determine which rows are to be deleted.)

TRUNCATE

Allows truncating a table using TRUNCATE.

REFERENCES

Allows creating a foreign key constraint referencing a table or specific columns of a table.

CREATE

For databases, this right allows creating schemas and publications, as well as installing trusted extensions in a specific database. For schemas, this right allows creating new objects in a given schema. To rename an existing object, one must be its owner and have this right for the schema containing it. For tablespaces, this right allows creating tables, indexes, and temporary files in a specific tablespace, and creating databases for which this space will be the default.

CONNECT

Allows connecting to a database. This right is checked when establishing a connection (in addition to the conditions defined in the pg_hba.conf configuration). For schemas, this right provides access to objects contained within them (assuming the necessary rights to access the objects themselves are also held). This right allows the subject to

"see" objects inside the schema. Without this permission, object names can still be discovered, for example, by querying system catalogs. Also, if this right is revoked, existing sessions may have statements for which object name visibility was performed earlier, so this right does not provide an absolutely reliable way to block access to objects. For sequences, this right allows using the `currval` and `nextval` functions. For types and domains, this right allows using the specified type or domain when creating tables, functions, or other schema objects. For foreign data wrappers, this right allows creating foreign server definitions that use them. For foreign servers, this right allows creating foreign tables that use them. Users having this right can also create, modify, or delete their own user mappings associated with a specific server.

20.2. Backup and Recovery

Backup is a fundamental aspect of working with a DBMS, aimed at ensuring data preservation and quick information recovery in case of unforeseen circumstances (hardware failure, accidental data deletion, cyberattack, etc.). The Digital Q.DataBase DBMS provides tools for reliable and efficient data backup and recovery.

Backup Procedure

1. Preparation

- Check for sufficient free space on the storage device.
- Archive old backup copies if they are no longer needed.

2. Creating a Backup

- Use the **qdb_dump** command for full backup.

Keys:

- -U username
- -h host
- -p port
- -F dump format: p (plain), c (custom), d (directory), t (tar)
- -f output file path
- -d database name
- --table=name dump only one table

- --schema=name only a specific schema
- -v verbose output
- --data-only only data, without schema
- --schema-only only schema, without data
- --inserts use INSERT instead of COPY

Sequence of actions for restoring data from a dump:

1. Stopping Active Processes

- Before restoring, ensure that no users are accessing the database. If necessary, stop web services or applications working with the database.

2. Restoring a Full Backup * If the backup was made in archive format, use the **pg_restore** command.

Keys:

- -U username
- -h host
- -p port
- -d target database
- -F format (usually not needed — determined automatically)
- -v verbose output
- -c drop objects before creating
- -C create database before restoring
- -j N parallel restore (N — number of threads)
- --list show dump contents
- --schema restore only the specified schema
- --table restore only the specified table

3. Restarting the Database

- After successful restoration, start the database and check its functionality by performing test queries and verifying data integrity.

20.3. Security and Privacy

Data security and privacy are among the most important aspects of working with the Digital Q.DataBase DBMS. Reliable protection of information from unauthorized access, theft, and destruction is a priority for companies and organizations. This section covers the main measures and recommendations for ensuring data security and privacy.

Organizational Measures

1. Security Policy:

- Develop and approve a security policy regulating data access, storage, and processing.

2. Staff Training:

- Conduct regular training and seminars for employees, informing them about security measures and responsibility for violations.

3. Physical Security:

- Organize physical security for premises where servers and equipment are located.

Technical Measures

1. Data Encryption:

- Encrypt data both at rest and during transmission over the network.

2. Access Control:

- Use authentication and authorization to control access to the database.

3. Monitoring and Auditing:

- Regularly maintain event logs and monitor suspicious activity.

4. Backup:

- Regularly back up data and store it in secure locations.

5. Antivirus Protection:

- Install antivirus programs and regularly update their signatures.

6. Firewalls and VPN:

- Use firewalls and VPN to protect the network from external threats.

Security Configuration Recommendations

1. Password Policy:

- Set strong passwords and change them regularly.

2. Principle of Least Privilege:

- Grant users the minimum set of privileges necessary to perform their duties.

3. SSL/TLS Encryption:

- Use SSL/TLS to encrypt traffic between clients and the server.

4. VPN and IPsec:

- Configure VPN and IPsec to protect networks and data transmission channels.

5. Fault Tolerance:

- Ensure hardware and communication channel redundancy to prevent data loss and operational disruptions.

21. Support Service

21.1. Support Service Contacts

21.2. Contacting Support Service

Contacting the support service is the first step towards quickly restoring operability and solving technical problems. It is important to prepare a request properly so that specialists can provide assistance promptly and efficiently. Following simple recommendations will reduce the time for identifying and eliminating a malfunction.

Algorithm for Contacting Support Service

1. Gathering Necessary Information:

- Product name and its version.
- Hardware model or server configuration.
- Detailed description of the problem and the conditions of its occurrence.
- Current product version.

2. Submitting a Request:

- Use the official support channel - email.
- Clearly formulate the subject of the request and briefly state the essence of the problem.

3. Detailed Problem Description:

- Compose a clear description of the situation, detailing the circumstances of the problem's occurrence.
- Attach error logs, screenshots of the error window, or other technical information confirming the request.

4. Waiting for Feedback:

- Specialists need time to study the problem and prepare a solution.

Rules for Formatting Error Reports

1. Clear Problem Description:

- Avoid vague formulations; clearly state the essence of the problem.

2. Detailing the Situation:

- Specify the exact date and time the error appeared.

3. Attaching Technical Details:

- If available — attach the error number and error message text.
- Take a screenshot of the error window or dialog panels.

4. Describing the Sequence of Actions:

- Reproduce the sequence of actions that led to the error.

5. Expected Result:

- Indicate what the result should have been without the error.

Example of Properly Formatted Error Message

Email Subject: Error in Digital Q.DataBase

Problem Description: This morning, when trying to run a sales report, the following error occurred: "ORA-00942: Table or view does not exist."

Technical Details:

- Date and Time: 2023-10-15, 09:30
- Environment: Local computer, OS Ubuntu Linux, DBMS version: 17.4
- Application version: 1.0.3
- Error Number: ORA-00942
- Error Message: "Table or view does not exist."

Previous Actions Taken:

1. Tried to log in again — the error repeated.
2. Checked the existence of the required table in the database — the table was indeed missing. Expected Result: The report should have loaded successfully and displayed up-to-date sales data.

Event Logs

Each time the service starts, a log file is created in the `/var/log/qdb/qdb_interpreter/` directory with the start time indicated in the file name. Files are created separately for the `qdbparse` and `qdbproxy` services.

22. Working with Data Migration Wizard

The Migration Wizard is a specialized tool developed by Diasoft, designed to migrate databases from various DBMS (including MS SQL, Oracle Database) to the domestic Digital Q.DataBase DBMS. The Migration Wizard ensures a smooth and reliable migration process, allowing users to easily transition to the new DBMS without the risk of data loss or performance degradation.

Main Functions of the Migration Wizard

1. Database Structure Migration:

- Migration of tables, views, indexes, constraints, and other database objects.

2. Data Migration:

- Full or selective migration of data from the source database to the new one.

3. Support for Multiple Migration Modes:

- Full migration (all objects and data).
- Structure migration (only objects).
- Data migration (only table population).
- Selective migration (specific objects and data).

4. User-Friendly Interface:

- Graphical interface (GUI) and console mode for the convenience of users with different skill levels.

Migration Process

1. Installing the Migration Wizard:

- Installation is performed by a simple installation wizard on Linux and Windows.

2. Launching the GUI Version of the Wizard:

- The program is launched by double-clicking or via the menu.

3. Selecting the Source DBMS:

- Specify the migration source (MS SQL Server, Oracle Database, PostgreSQL).

4. Specifying Connection Parameters:

- Enter credentials for connecting to the source database.

5. Selecting Object Types for Migration:

- You can choose to migrate the entire database, structure, data, or individual objects.

6. Selecting the Migration Target:

- A new database instance or an existing one.

7. Completing Migration:

- Click the "Migrate" button, wait for the operation to complete, and check the result.

Console Application

For experienced users, a console mode of operation for the Migration Wizard is provided. In the console version, you can more flexibly configure the migration process by creating YAML configurations for transferring data and structure.

23. Appendix 1. Extensions Included in Q.DataBase

No	Name	Description
1	adminpack	The extension provides several auxiliary functions that can be used by pgAdmin and other database administration and management tools, for example, for remote management of server log files.
2	amcheck	Extension for checking the logical integrity of index and table structures.
3	autoinc	Functions for auto-incrementing fields.
4	bloom	The extension provides an index access method based on Bloom filters.
5	btree_gin	The extension provides sample GIN operator classes that implement behavior similar to that of regular B-tree classes for data types int2, int4, int8, float4, float8, timestamp with time zone, timestamp without time zone, time with time zone, time without time zone, date, interval, oid, money, char, varchar, text, bytea, bit, varbit, macaddr, macaddr8, inet, cidr, uuid, name, bool, bpchar, and all enum types.
6	btree_gist	The btree_gist extension provides sample GiST operator classes that implement behavior similar to that of regular B-tree classes for data types int2, int4, int8, float4, float8, numeric, timestamp with time zone, timestamp without time zone, time with time zone, time without time zone, date, interval, oid, money, char, varchar, text, bytea, bit, varbit, macaddr, macaddr8, inet, cidr, uuid, and all enum types.
7	citext	The extension provides a data type for case-insensitive strings. It compares values by internally calling the lower function.
8	citus	Extension for Q.DataBase that provides scalability for Q.DataBase through transparent distribution and/or replication of tables across one or more Q.DataBase nodes.
9	cube	The extension implements the cube data type for representing multidimensional cubes.
10	dblink	Allows executing queries to data stored in other databases from the current session.
11	dict_int	The extension is an example of an additional dictionary template for full-text search. This dictionary was created for controlled indexing of integers; it allows indexing such numbers while avoiding excessive growth of the list of unique words, which significantly increases search speed.

No	Name	Description
12	dict_xsyn	The extension is an example of an additional dictionary template for full-text search. This dictionary replaces words with groups of their synonyms, allowing a word to be found by one of its synonyms.
13	earthdistance	The extension implements two different options for calculating orthodromic distance (distance between points on the Earth's surface).
14	fasttrun	The fasttrun extension provides a transactionally unsafe function for truncating temporary tables, preventing the bloating of the pg_class catalog.
15	file_fdw	The extension implements the file_fdw foreign data wrapper, which can be used to access data files in the server's file system or execute programs on the server and read their output.
16	fulleq	The fulleq extension provides an additional equality operator for compatibility with Microsoft SQL Server.
17	fuzzystrmatch	The extension contains several functions for calculating the similarity and distance between strings. A package for fuzzy string search.
18	hstore	The extension implements the hstore data type for storing key-value pairs within a single Q.DataBase value.
19	http	Extension for Q.DataBase that allows sending and processing HTTP requests directly from the database.
20	insert_username	The insert_username() function implements trigger code that saves the current username in a text field. This can be useful for tracking which user last modified a particular table row.
21	intagg	The extension provides an integer aggregator and enumerator.
22	intarray	The extension provides a number of useful functions and operators for working with arrays of integers without NULLs. It also supports indexed search for some of these operators.
23	isn	The extension provides data types for the following international product numbering standards: EAN13, UPC, ISBN (books), ISMN (music), and ISSN (serial numbers).
24	lo	The extension supports management of Large Objects (LO, sometimes BLOB, Binary Large Objects). It implements the lo data type and the lo_manage trigger.

No	Name	Description
25	ltree	The ltree extension is designed for storing and processing hierarchical data as labels in a tree structure. It also provides advanced tools for searching in such trees.
26	mchar	The mchar extension provides additional data types for compatibility with Microsoft SQL Server (MS SQL).
27	moddatetime	The moddatetime extension provides a trigger function that automatically updates the value of a timestamp field when a record in a table is updated.
28	old_snapshot	The extension allows reading information used in the implementation of old_snapshot_threshold.
29	oracle_fdw	The extension allows efficient integration of Q.DataBase and Oracle, providing transparent and secure access to data stored in Oracle from applications working with Q.DataBase.
30	pageinspect	The extension provides functions for examining database pages at a low level, which can be useful for debugging.
31	pg_buffercache	The extension makes it possible to understand what is happening in the shared buffer cache in real time.
32	pg_cron	The extension provides a convenient and flexible mechanism for automating regular maintenance tasks and data management within Q.DataBase, without the need to use external job scheduling systems.
33	pg_curl	Extension that allows executing HTTP requests using curl functions, which provide a wide range of capabilities for data exchange with external systems.
34	pg_freespacemap	The extension provides means for examining the Free Space Map (FSM). It implements the pg_freespace function, or rather, two overloaded functions. These functions show the value recorded in the free space map for a given page, or for all pages of a relation. Allows better understanding and management of disk space usage.
35	pg_hint_plan	Extension for Q.DataBase that allows specifying hints for the query planner. Hints can improve query performance by providing additional information on how the planner should process the query.
36	pg_prewarm	The extension provides a convenient way to load relation data into the operating system cache or the Q.DataBase buffer cache.

No	Name	Description
37	pg_query_state	Extension for Q.DataBase that provides information about the state of queries being executed in the database. This can be useful for monitoring and debugging queries, as well as for improving the performance and stability of Q.DataBase.
38	pg_squeeze	Extension for Q.DataBase, designed for automatically reducing the size of tables that have become bloated. This can help improve performance and reduce resource consumption.
39	pg_stat_kcache	The pg_stat_kcache extension is intended for collecting statistics about physical disk access performed by the database.
40	pg_stat_statements	The extension provides the ability to track execution statistics of all SQL queries across all databases on the server.
41	pg_surgery	The extension provides various functions for performing operations on corrupted relations.
42	pg_trgm	The extension provides functions and operators for determining the similarity of alphanumeric strings based on trigrams, as well as index operator classes that support fast search for similar strings.
43	pg_visibility	The extension makes it possible to examine the Visibility Map (VM) and page-level visibility information for a given table. It also provides functions for checking the integrity of the visibility map and forcing its rebuild.
44	pg_wait_sampling	The pg_wait_sampling extension is intended for collecting statistics about wait events in the database.
45	pg_walinspect	The extension provides SQL functions for viewing the write-ahead log at a low level. It works with a running database cluster and can be useful for debugging, analysis, reporting, or educational purposes.
46	pgaudit	Extension for Q.DataBase that provides detailed auditing of actions performed in the database.
47	pgcrypto	The extension provides cryptographic functions for Q.DataBase.
48	pgpool_adm	The pgpool_adm extension is designed for managing Pgpool-II via an SQL interface.
49	pgpool_recovery	Designed to support recovery functionality in Pgpool-II.
50	pgpool_regclass	The pgpool_regclass extension is designed for working with table names in Pgpool-II.

No	Name	Description
51	pgrowlocks	The extension provides a function that shows row locking information for a given table.
52	pgstattuple	The extension provides various functions for obtaining tuple (row)-level statistics in tables.
53	pldbgapi	Designed for debugging functions written in the PL/pgSQL language.
54	plpgsql	The extension allows performing syntax checks.
55	plpgsql_check	Extension for Q.DataBase designed for static analysis and checking of PL/pgSQL functions.
56	plugin_debugger	Extension for Q.DataBase designed to enable PL/pgSQL debugger functionality.
57	postgres_fdw	The extension provides the postgres_fdw foreign data wrapper, which can be used to access data located on external Q.DataBase servers.
58	qfixstat	Extension for the correct operation of the optimizer with p-tables; it is necessary to maintain p-table statistics in an optimal state.
59	qonline_analyze	The online_analyze extension provides a set of functions that immediately update statistics after INSERT, UPDATE, DELETE, or SELECT INTO operations on target tables.
60	qplansaver	The extension is designed to save query plans and use them bypassing the optimizer.
61	refint	Extension for implementing referential integrity between tables.
62	seg	The extension implements the seg data type for representing line segments or intervals of floating-point numbers. The seg type can express uncertainty in interval boundaries, allowing it to be used for representing laboratory measurements.
63	sslinfo	The extension provides information about the SSL certificate presented by the current client when connecting to Q.DataBase.
64	tablefunc	The extension contains a number of functions that return tables. Necessary for building pivot tables and reports from data stored in normalized form in the database.
65	tarantool_ext	Extension for processing data using Tarantool when working with Q.DataBase.
66	tcn	The extension offers a trigger function that notifies notification listeners about changes in the content of any table to which the trigger is attached.

No	Name	Description
67	tsm_system_rows	The extension is necessary for using the SYSTEM_ROWS sampling method in the SELECT statement with the TABLESAMPLE clause in Q.Database databases. This method allows obtaining a sample from a table, limiting it by the maximum number of rows specified in the argument.
68	tsm_system_time	The extension provides the SYSTEM_TIME sampling method, which can be used in the TABLESAMPLE clause of the SELECT command.
69	unaccent	The extension represents a text search dictionary that removes diacritical marks from lexemes.
70	uuid-oss	The extension provides functions for generating universally unique identifiers (UUIDs) using one of several standard algorithms. It also has functions that produce special UUID constants.
71	xml2	The extension provides functions for performing XPath queries and XSLT transformations.