db2utils Documentation

Release 0.2

Dave Hughes

Mar 05, 2018

Contents

db2utils is a collection of utility routines for IBM DB2 for Linux/UNIX/Windows (DB2 for LUW) which I have developed over several years as a DBA to make my duties a little easier.

Downloads are available from GitHub which also hosts the source code, and the bug tracker. Documentation is hosted by ReadTheDocs, and includes requirements, installation instructions, and an extensive reference section. The project is licensed under the MIT license.

Please feel free to contact me with questions, suggestions or patches!

CHAPTER 1

Table of Contents

1.1 Introduction

db2utils is a collection of utility routines for IBM DB2 for Linux/UNIX/Windows (DB2 for LUW) which I have developed over several years as a DBA to make my duties a little easier. The package has been tested on DB2 9.7, 10.1, and 10.5 under Linux (and previously with DB2 9.5 under Linux, although I cannot currently test with this version).

The utilities cover a range of topics including:

- Manipulation of user authorizations including copy all authorizations from one user to another
- Numerous date/time manipulation functions including a table-function for generating arbitrary date ranges
- Management of temporal data including automatic construction of effective-expiry-style history tables, the triggers to maintain them, and various views of historical data
- Perl-compatible regular expression functions including searching, substitution and splitting
- Automatic construction of exception tables (and analysis views) as used by the built-in LOAD utility and the SET INTEGRITY command
- Utilities for easy reconstruction of invalidated views and triggers (rather redundant as of 9.7, but probably still useful on 9.5)
- Utility functions which ease the construction of procedures which generate SQL (e.g. string and identifier quoting, construction of comma-separated column lists)

All functions and procedures are reasonably well documented in these pages, in comments in the source files, and with COMMENT ON statements within the database. Per-module and suite-wide roles are also defined to permit easy management of which users have access to which routines.

A simple installation procedure is provided for Linux/UNIX users, but Windows support is on an “if you can get it working” basis: I don’t have any DB2 for Windows installations to play with and I’ve no idea how one compiles external C-based UDFs on Windows.

1.2 Requirements

Obviously you'll want a relatively recent installation of DB2 for Linux/UNIX/Windows. Currently, the package has been tested on the following versions and platforms:

- DB2 9.5 for Linux (64-bit)
- DB2 9.7 for Linux (64-bit)
- DB2 10.1 for Linux (64-bit)
- DB2 10.5 for Linux (64-bit)

1.2.1 Linux

As db2utils includes C-based external routines, a C compiler is required (`gcc` is the only one I've tested thus far). GNU `make` is used to ease the installation process, and GNU `awk` is used as part of the test script. The PCRE library and headers are required by the pcre functions. All these pre-requisites can be installed quite easily using your distro's package manager. Instructions for specific distros are below:

Ubuntu `$ sudo apt-get install build-essential gawk libpcre3 libpcre3-dev`

Gentoo (with Portage) (you almost certainly already have all pre-requisites installed, but if not):

```
$ sudo emerge sys-apps/gawk sys-devel/make sys-devel/gcc dev-libs/libpcre
```

Gentoo (with Paludis) (you almost certainly already have all pre-requisites installed, but if not):

```
$ sudo cave resolve -x sys-apps/gawk sys-devel/make sys-devel/gcc
dev-libs/libpcre
```

1.2.2 Windows

What compiler is required for building C-based external routines? How does one install and configure it? How does one execute Makefiles on Windows? Can Cygwin/MingW be used for any of this? If anyone wants to figure this all out, be my guest...

1.3 Downloads

The releases of the library in reverse chronological order are listed below along with download links:

- Release 0.1 (2013-08-16)
- Release 0.2 (2017-09-01)

1.3.1 Development

If you wish to develop db2utils itself, you are recommended to do so from a clone of the GitHub repository which can be obtained like so:

```
$ git clone https://github.com/waveform-computing/db2utils.git
```

Alternatively, fork the repository on GitHub, develop on your copy and submit a pull request.

1.4 Installation

First, make sure you've installed the *Requirements*, then following the instructions in the section for your platform below.

1.4.1 Linux

Log on as a user which has **SYSADM** authority for the DB2 instance you wish to install under (commonly this is *db2inst1*), and ensure the **db2profile** for the target DB2 instance has been sourced (this is the usually the case with the *db2inst1* user):

```
$ su - db2inst1
$ source ~db2inst1/sqllib/db2profile
```

Extract the archive you downloaded, and change to the directory it creates:

```
$ tar -xzf db2utils-release-0.1.tar.gz
$ cd db2utils-release-0.1
```

Edit the two variables **DBNAME** and **SCHEMANAME** at the top of the Makefile:

```
$ ${EDITOR} Makefile
```

These indicate the database into which to install everything and the schema under which to place all objects. Finally, use the included Makefile to make the "install" target:

```
$ make install
```

This will compile the external pcre UDFs library, install it in the instance identified by the **DB2INSTANCE** environment variable (which is set by **db2profile**), then connect to the database identified by **DBNAME** and install everything under the schema specified by **SCHEMANAME**.

If you wish to see the SQL that would be executed without actually executing it (if, for example, you wish to edit it before hand) you can create it with the following target:

```
$ make install.sql
```

If you wish to uninstall everything from the database, simply make the "uninstall" target:

```
$ make uninstall
```

There is also a target which attempts to test the implementation of various functions and procedures by using the functions in the assert.sql module. This can be run with the "test" target:

```
$ make test
```

The test suite is currently rather crude. Any error immediately stops the test suite to allow examination. If the test suite runs to the end, this indicates success.

1.4.2 Windows

Anyone want to figure this out?

1.5 First Steps

The package installs a variety of functions and procedures under the *UTILS* schema by default. The functions are divided into modules and each module defines at least two roles which can be used to grant access to the functions of that module. The roles are always named **UTILS_module_USER** and **UTILS_module_ADMIN**. For example, the auth.sql defines **UTILS_AUTH_USER** and **UTILS_AUTH_ADMIN**. The **UTILS_AUTH_USER** role has the ability to execute all procedures and functions within the module. The **UTILS_AUTH_ADMIN** role also has these execute privileges and in addition has the ability to grant the **UTILS_AUTH_USER** role to other users and roles.

In addition to the per-module roles, there are also a couple of other roles: **UTILS_USER** and **UTILS_ADMIN**. **UTILS_USER** holds all the per-module user roles, while **UTILS_ADMIN** holds all the per-module administrative roles so if you wish to grant access to the entire suite, simply grant one of these two roles. Naturally, **UTILS_ADMIN** also holds the ability to grant **UTILS_USER**, and in addition has **CREATEIN**, **DROPIN**, and **ALTERIN** privileges on the target schema.

Hence, after installing the package your first step will likely be to assign some roles to other roles. For example, let's assume you have a role called **DEVELOPERS** who should have access to the entire suite of functions in db2utils. Let's also assume there's a role for ordinary users called **QUERY_USERS** who should only have access to the enhanced date-time functions in the date_time.sql module. Finally, there's a role for administrative users called **ADMINS** who should have administrative control over the package. In this case, after installation you would do the following:

```
$ db2 GRANT ROLE UTILS_ADMIN TO ROLE ADMINS WITH ADMIN OPTION
$ db2 GRANT ROLE UTILS_USER TO ROLE DEVELOPERS
$ db2 GRANT ROLE UTILS_DATE_TIME_USER TO ROLE QUERY_USERS
```

In order to provide easier access to the functions and procedures in the package you will likely want to alter your function search path:

```
$ db2 SET PATH SYSTEM PATH, USER, UTILS
```

If you use the utilities regularly you may wish to construct a small script, alias, or function for connecting to your database and setting the function search path automatically. For example, in my *.bashrc* I have:

```
sample() {
    # Ensure the correct instance is active
    db2 TERMINATE
    source ~db2inst1/sqllib/db2profile
    # Connect to the database and set up the environment
    db2 CONNECT TO SAMPLE
    db2 SET PATH SYSTEM PATH, USER, UTILS
    db2 SET SCHEMA MAIN
}
```

1.6 Modules Overview

The routines are divided into modules roughly by topic:

assert.sql Includes a set of procedures and functions for performing assertion tests against the framework.

auth.sql Includes a set of procedures for managing authorizations, including the ability to copy, remove, and move all authorizations for a given ID, and save and restore authorizations on relations.

corrections.sql In the databases I work with there is frequently a need to correct data sourced from other databases, typically names of entities which weren't "neat enough" for reporting purposes. We accomplished this by having an "original name" column, a "corrected name" column, and finally the name column itself would be a generated column coalescing the two together. Only those names that required correction would have a value in the "corrected name" column, and a trigger on the table would ensure that corrections would be wiped in the event that the "original name" changed (on the assumption that the correction would need changing). This module contains procedures for creating the trigger.

date_time.sql Contains numerous functions for handling `DATE`, `TIME` and `TIMESTAMP` values including calculating the start and end dates of years, quarters, and months, calculating the next or previous of a particular day of the week (Monday, Tuesday, etc.), formatting timestamps with strftime() style templates, and a table function for generating a range of dates.

drop_schema.sql Contains a procedure for dropping all objects in a schema, and the schema itself. This is redundant as of DB2 9.5 which incldues ADMIN_DROP_SCHEMA, but the syntax is a bit easier for this one as it doesn't rely on a table to report errors (if something goes wrong it just fails and throws an SQL error).

evolve.sql Contains procedures which make schema evolution (changing views and such like) a bit easier. This is redundant as of DB2 9.7 which includes much better schema evolution capabilities (deferred revalidation), but may still be useful for people on earlier versions. The routines include the ability to save and restore view definitions, including authorizations, and routines for easily recreating invalid views and triggers from their definitions in the system catalog.

exceptions.sql Contains procedures for creating exception tables and views. Exception tables have the same structure as a base table but with two extra columns for reporting errors that occur during a `LOAD` or `SET INTEGRITY` command. Exception views translate the (rather cryptic) extra columns in exception tables into human readable information.

export_load.sql Contains functions for generating `EXPORT` and `LOAD` commands for tables or schemas of tables. These can be used to easily generate CLP scripts which mimic the function of `db2move`, but with all the filtering capabilities of SQL (i.e. you could limit the scope with more fidelity than just specifying a schema), and with functionality to cope with `IDENTITY` and `GENERATED` columns properly (which `db2move` has problems with).

history.sql Contains procedures for creating "history" tables, triggers, and views. History tables track the changes to a base table over time. Triggers on the base table are used to populate the history table. Views on the history table can be created to provide different perspectives on the history data (snapshots over time, explicit lists of what changes occured, etc).

log.sql Contains a table and a procedure for logging administrative alerts and information. This module isn't complete yet; plenty of functionality I'd like to implement when I get the time. . .

merge.sql Defines a set of procedures for automatically generating `INSERT`, `DELETE`, and `MERGE` statements with the intention of bulk-transferring data between similarly structured tables.

pcre.sql Defines a set of functions providing PCRE (Perl Compatible Regular Expression) search, split and substitution functionality. The functions are implemented in a C library the source for which is in the pcre/ sub-directory.

sql.sql Contains a couple of simple functions for escaping strings and identifiers in SQL. Used by numerous of the modules for generating SQL dynamically.

toggle_triggers.sql Contains procedures for easily disabling and enabling triggers, including specific triggers or all triggers on a given table.

unicode.sql Defines functions for cleaning up Unicode strings, in particular those using the common UTF-8 encoding scheme. The functions are implemented in a C library the source for which is in the unicode/ sub-directory.

1.7 Reference

Each procedure and function is documented below (or will be once I get the time!). Click on a routine name to view the documentation:

1.7.1 Functions

ASSERT_EQUALS scalar function

Signals ASSERT_FAILED_STATE if A does not equal B.

Prototypes

```
ASSERT_EQUALS(A INTEGER, B INTEGER)
ASSERT_EQUALS(A DOUBLE, B DOUBLE)
ASSERT_EQUALS(A TIMESTAMP, B TIMESTAMP)
ASSERT_EQUALS(A TIME, B TIME)
ASSERT_EQUALS(A VARCHAR(4000), B VARCHAR(4000))

RETURNS INTEGER
```

Description

Raises the ASSERT_FAILED_STATE state if **A** does not equal **B**. The function is overloaded for most common types and generally should not need CASTs for usage. The return value in the case that the values are equal is arbitrary.

Parameters

A The first value to compare

B The value to compare to A

Examples

Test that TIMESTAMP is constant within an expression:

```
VALUES ASSERT_EQUALS(CURRENT TIMESTAMP, CURRENT TIMESTAMP);
```

```
1
-----------
          0
```

Test an obvious failure:

```
VALUES ASSERT_EQUALS(1, 2);
```

```
1
-----------
SQL0438N  Application raised error or warning with diagnostic text: "1 does
not equal 2".  SQLSTATE=90001
```

See Also

- Source code

- *ASSERT_NOT_EQUALS scalar function*
- *ASSERT_IS_NULL scalar function*
- *ASSERT_IS_NOT_NULL scalar function*
- ASSERT_FAILED_STATE

ASSERT_IS_NOT_NULL scalar function

Signals ASSERT_FAILED_STATE if A is NULL.

Prototypes

```
ASSERT_IS_NOT_NULL(A INTEGER)
ASSERT_IS_NOT_NULL(A DOUBLE)
ASSERT_IS_NOT_NULL(A TIMESTAMP)
ASSERT_IS_NOT_NULL(A TIME)
ASSERT_IS_NOT_NULL(A VARCHAR(4000))

RETURNS INTEGER
```

Description

Raises the ASSERT_FAILED_STATE state if **A** is NULL. The function is overloaded for most common types and generally should not need CASTs for usage. The return value in the case that the value is not NULL is arbitrary.

Parameters

A The value to check for NULL.

Examples

Test an obvious tautology:

```
VALUES ASSERT_IS_NOT_NULL(1);
```

```
1
-----------
          0
```

Test that the *DATE scalar function* function returns NULL on NULL input:

```
VALUES ASSERT_IS_NOT_NULL(DATE(2000, 1, NULL));
```

```
1
-----------
SQL0438N  Application raised error or warning with diagnostic text: "Value
is NULL".  SQLSTATE=90001
```

See Also

- Source code
- *ASSERT_IS_NULL scalar function*
- *ASSERT_NOT_EQUALS scalar function*
- *ASSERT_EQUALS scalar function*
- ASSERT_FAILED_STATE

ASSERT_IS_NULL scalar function

Signals ASSERT_FAILED_STATE if A is a non-NULL value.

Prototypes

```
ASSERT_IS_NULL(A INTEGER)
ASSERT_IS_NULL(A DOUBLE)
ASSERT_IS_NULL(A TIMESTAMP)
ASSERT_IS_NULL(A TIME)
ASSERT_IS_NULL(A VARCHAR(4000))

RETURNS INTEGER
```

Description

Raises the ASSERT_FAILED_STATE state if **A** is not NULL. The function is overloaded for most common types and generally should not need CASTs for usage. The return value in the case that the value is NULL is arbitrary.

Parameters

A The value to check for NULL.

Examples

Test an obvious failure:

```
VALUES ASSERT_IS_NULL(1);
```

```
1
-----------
SQL0438N  Application raised error or warning with diagnostic text: "1 is
non-NULL".  SQLSTATE=90001
```

Test that the *DATE scalar function* function returns NULL on NULL input:

```
VALUES ASSERT_IS_NULL(DATE(2000, 1, NULL));
```

```
1
-----------
          0
```

See Also

- Source code
- *ASSERT_IS_NOT_NULL scalar function*
- *ASSERT_NOT_EQUALS scalar function*
- *ASSERT_EQUALS scalar function*
- ASSERT_FAILED_STATE

ASSERT_NOT_EQUALS scalar function

Signals ASSERT_FAILED_STATE if A equals B.

Prototypes

```
ASSERT_NOT_EQUALS(A INTEGER, B INTEGER)
ASSERT_NOT_EQUALS(A DOUBLE, B DOUBLE)
ASSERT_NOT_EQUALS(A TIMESTAMP, B TIMESTAMP)
ASSERT_NOT_EQUALS(A TIME, B TIME)
ASSERT_NOT_EQUALS(A VARCHAR(4000), B VARCHAR(4000))

RETURNS INTEGER
```

Description

Raises the ASSERT_FAILED_STATE state if **A** equals **B**. The function is overloaded for most common types and generally should not need CASTs for usage. The return value in the case that the values aren't equal is arbitrary.

Parameters

A The first value to compare

B The value to compare to A

Examples

Test that the LEFT function works:

```
VALUES ASSERT_NOT_EQUALS('AAA', LEFT('AAA', 1));
```

```
1
-----------
          0
```

Test an obvious failure:

```
VALUES ASSERT_NOT_EQUALS(1, 1);
```

```
1
-----------
SQL0438N  Application raised error or warning with diagnostic text: "
Values are both 1".  SQLSTATE=90001
```

See Also

- Source code
- *ASSERT_EQUALS scalar function*
- *ASSERT_IS_NULL scalar function*
- *ASSERT_IS_NOT_NULL scalar function*
- ASSERT_FAILED_STATE

AUTH_DIFF table function

Utility table function which returns the difference between the authorities held by two names.

Prototypes

```
AUTH_DIFF(SOURCE VARCHAR(128), SOURCE_TYPE VARCHAR(1), DEST VARCHAR(128), DEST_TYPE␣
↪VARCHAR(1), INCLUDE_COLUMNS VARCHAR(1), INCLUDE_PERSONAL VARCHAR(1))
AUTH_DIFF(SOURCE VARCHAR(128), DEST VARCHAR(128), INCLUDE_COLUMNS VARCHAR(1), INCLUDE_
↪PERSONAL VARCHAR(1))
AUTH_DIFF(SOURCE VARCHAR(128), DEST VARCHAR(128), INCLUDE_COLUMNS VARCHAR(1))

RETURNS TABLE(
  OBJECT_TYPE VARCHAR(18),
  OBJECT_ID VARCHAR(262),
  AUTH VARCHAR(140),
  SUFFIX VARCHAR(20),
  LEVEL SMALLINT
)
```

Description

This utility function determines the difference in authorizations held by two different entities (as determined by *AUTHS_HELD table function*). Essentially it takes the authorizations of the SOURCE entity and "subtracts" the authorizations of the **DEST** entity, the result being the authorizations that need to be granted to **DEST** to give it the same level of access as **SOURCE**. This is used in the definition of the *COPY_AUTH procedure* routine.

Parameters

SOURCE The name to check for existing authorizations.

SOURCE_TYPE The type of the **SOURCE** parameter. Specify `'U'`, `'G'`, or `'R'` for User, Group or Role respectively. If this parameter is omitted, the type will be determined by the *AUTH_TYPE scalar function* function.

DEST The intended destination for the authorizations held by **SOURCE**.

DEST_TYPE The type of the **DEST** parameter. Takes the same values as **SOURCE_TYPE**. If omitted, the type will be determined by the *AUTH_TYPE scalar function* function.

INCLUDE_COLUMNS If this parameter is `'Y'`, column level authorizations will be included.

INCLUDE_PERSONAL If this parameter is `'Y'`, and **SOURCE** identifies a user, then authorizations for the source user's personal schema will be included in the result. This parameter defaults to `'N'` when omitted.

Returns

See the *AUTHS_HELD table function* documentation for a description of the columns of the returned table (this routine is essentially a "subtraction" of two AUTHS_HELD calls hence the output structure is identical).

Examples

Show the authorizations directly granted to the *DB2INST1* user which the currently logged on user does not possess.

```
SELECT * FROM TABLE(AUTH_DIFF('DB2INST1', USER, 'N'));
```

```
OBJECT_TYPE OBJECT_ID                  AUTH        SUFFIX               LEVEL
----------- -------------------------- ----------- -------------------- ------
PACKAGE     NULLID.POLYH03             CONTROL                               0
INDEX       SYSTOOLS.ATM_UNIQ          CONTROL                               0
INDEX       SYSTOOLS.HI_OBJ_UNIQ       CONTROL                               0
TABLE       SYSTOOLS.HMON_ATM_INFO     CONTROL                               0
TABLE       SYSTOOLS.HMON_COLLECTION   CONTROL                               0
TABLE       SYSTOOLS.POLICY            CONTROL                               0
INDEX       SYSTOOLS.POLICY_UNQ        CONTROL                               0
TABLE       SYSTOOLS.HMON_ATM_INFO     ALTER       WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_COLLECTION   ALTER       WITH GRANT OPTION         1
TABLE       SYSTOOLS.POLICY            ALTER       WITH GRANT OPTION         1
PACKAGE     NULLID.POLYH03             BIND        WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_ATM_INFO     DELETE      WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_COLLECTION   DELETE      WITH GRANT OPTION         1
TABLE       SYSTOOLS.POLICY            DELETE      WITH GRANT OPTION         1
PACKAGE     NULLID.POLYH03             EXECUTE     WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_ATM_INFO     INDEX       WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_COLLECTION   INDEX       WITH GRANT OPTION         1
TABLE       SYSTOOLS.POLICY            INDEX       WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_ATM_INFO     INSERT      WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_COLLECTION   INSERT      WITH GRANT OPTION         1
TABLE       SYSTOOLS.POLICY            INSERT      WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_ATM_INFO     REFERENCES  WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_COLLECTION   REFERENCES  WITH GRANT OPTION         1
TABLE       SYSTOOLS.POLICY            REFERENCES  WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_ATM_INFO     SELECT      WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_COLLECTION   SELECT      WITH GRANT OPTION         1
TABLE       SYSTOOLS.POLICY            SELECT      WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_ATM_INFO     UPDATE      WITH GRANT OPTION         1
TABLE       SYSTOOLS.HMON_COLLECTION   UPDATE      WITH GRANT OPTION         1
TABLE       SYSTOOLS.POLICY            UPDATE      WITH GRANT OPTION         1
```

```
TABLESPACE  SYSTOOLSPACE                USE         WITH GRANT OPTION         1
TABLESPACE  SYSTOOLSTMPSPACE            USE         WITH GRANT OPTION         1
```

See Also

- Source code
- *AUTH_TYPE scalar function*
- *AUTHS_HELD table function*
- *COPY_AUTH procedure*
- *MOVE_AUTH procedure*
- *REMOVE_AUTH procedure*

AUTH_TYPE scalar function

Utility routine used by other routines to determine the type of an authorization name when it isn't explicitly given.

Prototypes

```
AUTH_TYPE(AUTH_NAME VARCHAR(128))
RETURNS VARCHAR(1)
```

Description

This is a utility function used by the *COPY_AUTH procedure* procedure, and other associated procedures. Given an authorization name, this scalar function returns `'U'`, `'G'`, or `'R'` to indicate that **AUTH_NAME** is a user, group, or role respectively (based on the content of the system catalog tables). If the name is undefined, `'U'` is returned, unless the name is *PUBLIC* in which case `'G'` is returned (for consistency with the catalog tables). If the name represents multiple authorization types, SQLSTATE 21000 is raised.

Parameters

AUTH_NAME The authorization name to test for type.

Examples

Show the type of the *PUBLIC* authorization name.

```
VALUES AUTH_TYPE('PUBLIC');
```

```
1
-
G
```

Show the type of the authorization name of the currently logged on user.

```
VALUES AUTH_TYPE(CURRENT USER);
```

```
1
-
U
```

See Also

- Source code
- *AUTHS_HELD table function*
- *AUTH_DIFF table function*
- *COPY_AUTH procedure*
- *MOVE_AUTH procedure*
- *REMOVE_AUTH procedure*

AUTHS_HELD table function

Utility table function which returns all the authorizations held by a specific name.

Prototypes

```
AUTHS_HELD(AUTH_NAME VARCHAR(128), AUTH_TYPE VARCHAR(1), INCLUDE_COLUMNS VARCHAR(1),␣
↪INCLUDE_PERSONAL VARCHAR(1))
AUTHS_HELD(AUTH_NAME VARCHAR(128), INCLUDE_COLUMNS VARCHAR(1), INCLUDE_PERSONAL␣
↪VARCHAR(1))
AUTHS_HELD(AUTH_NAME VARCHAR(128), INCLUDE_COLUMNS VARCHAR(1))

RETURNS TABLE(
    OBJECT_TYPE VARCHAR(18),
    OBJECT_ID VARCHAR(262),
    AUTH VARCHAR(140),
    SUFFIX VARCHAR(20),
    LEVEL SMALLINT
)
```

Description

This is a utility function used by *COPY_AUTH procedure*, and other associated procedures, below. Given an authorization name and type, and a couple of flags, this table function returns the details of all the authorizations held by that name. The information returned is sufficient for comparison of authorizations and generation of GRANT/REVOKE statements.

Parameters

AUTH_NAME The authorization name to query authorizations for.

AUTH_TYPE The type of the authorization name. Use `'U'` for users, `'G'` for groups, or `'R'` for roles. If this parameter is omitted the type will be determined by calling the *AUTH_TYPE scalar function* function.

INCLUDE_COLUMNS If this is `'Y'` then include column-level authorizations for relations (tables, views, etc). This is useful when generating REVOKE statements from the result (as column level authorizations cannot be revoked directly in DB2).

INCLUDE_PERSONAL This parameter controls whether, in the case where **AUTH_NAME** refers to a user (as opposed to a group or role), authorizations associated with the user's personal schema are included in the result. If set to `'Y'`, personal schema authorizations are included. Defaults to `'N'` if omitted.

Returns

The function returns one row per authorization found in the system catalogs for the specified authorization name. Each row contains the following columns:

OBJECT_TYPE This column typically contains a string indicating the type of object identified by the *OBJECT_ID* column. However, given that this routine's primary purpose is to aid in the generation of GRANT and REVOKE statements, and given the inconsistencies in the numerous GRANT and REVOKE syntaxes employed by DB2, this column is blank for certain object types (roles and security labels), and misleading for others (e.g. `'TABLE'` is returned for all relation types including views).

OBJECT_ID The identifier of the object the authorization was granted upon. This will be the schema-qualified name for those objects that reside in a schema, and will be properly quoted (if required) for inclusion in generated SQL.

AUTH The name of the authority granted upon the *OBJECT_ID*. For example, if *OBJECT_TYPE* is `'DATABASE'` this might be `'BINDADD'` or `'IMPLICIT_SCHEMA'`. Alternatively, if *OBJECT_TYPE* is `'TABLE'` this could be `'SELECT'` or `'ALTER'`. As the function's purpose is to aid in generating GRANT and REVOKE statements, the name of the authority is always modelled after what would be used in the syntax of these statements.

SUFFIX Several authorizations can be granted with additional permissions. For example in the case of tables, SELECT authority can be granted with or without the GRANT OPTION (the ability for the grantee to pass on the authority to others), while roles can be granted with or without the ADMIN OPTION (the ability for the grantee to grant the role to others). If such a suffix is associated with the authority, this column will contain the syntax required to grant that option.

LEVEL This is a numeric indicator of the "level" of a grant. As discussed in the description of the SUFFIX column above, authorities can sometimes be granted with additional permissions. In such cases this column is a numeric indication of the presence of additional permissions (for example, a simple SELECT grant would be represented by 0, with SELECT WITH GRANT OPTION would be 1). This is used by *COPY_AUTH procedure* when comparing two sets of authorities to determine whether a grant needs "upgrading" (say from SELECT to SELECT WITH GRANT OPTION).

Examples

Show the authorizations held by the *PUBLIC* group, limiting the results to 10 authorizations per object type (otherwise the results are huge!).

```
WITH T AS (
  SELECT
    ROW_NUMBER() OVER (
      PARTITION BY OBJECT_TYPE
      ORDER BY OBJECT_ID
    ) AS ROWNUM,
    T.*
  FROM
    TABLE (AUTHS_HELD('PUBLIC', 'N')) AS T
)
SELECT
  T.OBJECT_TYPE,
  T.OBJECT_ID,
  T.AUTH,
  T.SUFFIX,
  T.LEVEL
FROM
  T
WHERE
  T.ROWNUM <= 10
```

```
OBJECT_TYPE          OBJECT_ID                                           AUTH                    ␣
↪SUFFIX                  LEVEL
------------------ ------------------------------------------ -------------------- ---
↪----------------- ------
DATABASE                                                                 BINDADD                    ␣
↪                         0
DATABASE                                                                 CONNECT                    ␣
↪                         0
DATABASE                                                                 CREATETAB                  ␣
↪                         0
DATABASE                                                                 IMPLICIT_SCHEMA            ␣
↪                         0
PACKAGE              NULLID.AOTMH00                                      BIND                       ␣
↪                         0
PACKAGE              NULLID.AOTMH00                                      EXECUTE                    ␣
↪                         0
PACKAGE              NULLID.ATSH04                                       BIND                       ␣
↪                         0
PACKAGE              NULLID.ATSH04                                       EXECUTE                    ␣
↪                         0
PACKAGE              NULLID.DB2XDBMI                                     BIND                       ␣
↪                         0
PACKAGE              NULLID.DB2XDBMI                                     EXECUTE                    ␣
↪                         0
PACKAGE              NULLID.PRINTSG                                      BIND                       ␣
↪                         0
PACKAGE              NULLID.PRINTSG                                      EXECUTE                    ␣
↪                         0
PACKAGE              NULLID.REVALH03                                     BIND                       ␣
↪                         0
PACKAGE              NULLID.REVALH03                                     EXECUTE                    ␣
↪                         0
PROCEDURE            SYSIBM.*                                            EXECUTE                    ␣
↪                         0
SCHEMA               DAVE                                                CREATEIN                   ␣
↪                         0
SCHEMA               NULLID                                              CREATEIN                   ␣
↪                         0
SCHEMA               SQLJ                                                CREATEIN                   ␣
↪                         0
SCHEMA               SYSPUBLIC                                           CREATEIN                   ␣
↪                         0
SCHEMA               SYSPUBLIC                                           DROPIN                     ␣
↪                         0
```

```
SCHEMA             SYSTOOLS                                  CREATEIN                ␣
↪                       0
SCHEMA             UTILS                                     CREATEIN                ␣
↪                       0
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_CONTACTGROUPS           EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_CONTACTS                EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_DBP_MEM_USAGE           EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_DBP_MEM_USAGE_AP        EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_INDEX_COMPRESS_INFO     EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_INDEX_INFO              EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_MSGS                    EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO       EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_TAB_COMPRESS_INFO_V97   EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC FUNCTION  SYSPROC.ADMIN_GET_TAB_INFO                EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.DB2_INSTALL_JAR                      EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.DB2_INSTALL_JAR2                     EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.DB2_REPLACE_JAR                      EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.DB2_UPDATEJARINFO                    EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.RECOVERJAR                           EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.REFRESH_CLASSES                      EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.REMOVE_JAR                           EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SQLJ.REMOVE_JAR2                          EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SYSFUN.GET_SAR                            EXECUTE              ␣
↪WITH GRANT OPTION         1
SPECIFIC PROCEDURE SYSFUN.GET_SAR4PARM                       EXECUTE              ␣
↪WITH GRANT OPTION         1
TABLE              SYSCAT.ATTRIBUTES                         SELECT                  ␣
↪                       0
TABLE              SYSCAT.AUDITPOLICIES                      SELECT                  ␣
↪                       0
TABLE              SYSCAT.AUDITUSE                           SELECT                  ␣
↪                       0
TABLE              SYSCAT.BUFFERPOOLDBPARTITIONS             SELECT                  ␣
↪                       0
TABLE              SYSCAT.BUFFERPOOLNODES                    SELECT                  ␣
↪                       0
TABLE              SYSCAT.BUFFERPOOLS                        SELECT                  ␣
↪                       0
TABLE              SYSCAT.CASTFUNCTIONS                      SELECT                  ␣
↪                       0
```

```
TABLE               SYSCAT.CHECKS                               SELECT                    ␣
↪           0
TABLE               SYSCAT.COLAUTH                              SELECT                    ␣
↪           0
TABLE               SYSCAT.COLCHECKS                            SELECT                    ␣
↪           0
TABLESPACE          SYSTOOLSTMPSPACE                            USE                       ␣
↪           0
TABLESPACE          USERSPACE1                                  USE                       ␣
↪           0
WORKLOAD            SYSDEFAULTUSERWORKLOAD                      USAGE                     ␣
↪           0
```

See Also

- Source code
- *AUTH_TYPE scalar function*
- *AUTH_DIFF table function*
- *COPY_AUTH procedure*
- *MOVE_AUTH procedure*
- *REMOVE_AUTH procedure*

DATE scalar function

Returns a DATE constructed from the specified year, month and day (or day of year).

Prototypes

```
DATE(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER)
DATE(AYEAR INTEGER, ADOY INTEGER)

RETURNS DATE
```

Description

Returns the DATE value with the components specified by **AYEAR**, **AMONTH** and **ADAY**, or alternatively **AYEAR** and **ADOY** the latter of which is the day of year to construct a DATE for.

Parameters

AYEAR Specifies the year component of the resulting date.

AMONTH If provided, specifies the month component of the resulting date.

ADAY If provided, specifies the day (of month) component of the resulting date.

ADOY If provided, specifies the day of year from which the month and day components of the resulting date will be calculated (the first day of a year is numbered 1).

Examples

Construct a date for first day in February, 2010:

```
VALUES DATE(2010, 2, 1);
```

```
1
----------
2010-02-01
```

Construct a date for the 180th day of 2009:

```
VALUES DATE(2009, 180);
```

```
1
----------
2009-06-29
```

See Also

- Source code
- *TIME scalar function*
- DATE (built-in function)

DATE_RANGE table function

Returns a table of DATEs from **START** to **FINISH** (inclusive), incrementing by **STEP** with each row (where **STEP** is an 8 digit duration formatted as YYYYMMDD, which defaults to 1 day).

Prototypes

```
DATE_RANGE(START DATE, FINISH DATE, STEP DECIMAL(8, 0))
DATE_RANGE(START DATE, FINISH TIMESTAMP, STEP DECIMAL(8, 0))
DATE_RANGE(START TIMESTAMP, FINISH DATE, STEP DECIMAL(8, 0))
DATE_RANGE(START TIMESTAMP, FINISH TIMESTAMP, STEP DECIMAL(8, 0))
DATE_RANGE(START DATE, FINISH VARCHAR(26), STEP DECIMAL(8, 0))
DATE_RANGE(START VARCHAR(26), FINISH DATE, STEP DECIMAL(8, 0))
DATE_RANGE(START VARCHAR(26), FINISH VARCHAR(26), STEP DECIMAL(8, 0))
DATE_RANGE(START TIMESTAMP, FINISH VARCHAR(26), STEP DECIMAL(8, 0))
DATE_RANGE(START VARCHAR(26), FINISH TIMESTAMP, STEP DECIMAL(8, 0))
DATE_RANGE(START DATE, FINISH DATE)
DATE_RANGE(START DATE, FINISH TIMESTAMP)
DATE_RANGE(START TIMESTAMP, FINISH DATE)
DATE_RANGE(START TIMESTAMP, FINSIH TIMESTAMP)
DATE_RANGE(START DATE, FINISH VARCHAR(26))
DATE_RANGE(START VARCHAR(26), FINISH DATE)
DATE_RANGE(START VARCHAR(26), FINISH VARCHAR(26))
DATE_RANGE(START TIMESTAMP, FINISH VARCHAR(26))
DATE_RANGE(START VARCHAR(26), FINISH TIMESTAMP)

RETURNS TABLE(
  D DATE
)
```

Description

DATE_RANGE generates a range of dates from **START** to **FINISH** inclusive, advancing in increments given by the date duration **STEP**. Date durations are DECIMAL(8,0) values structured as YYYYMMDD (in DB2 they are typically derived as the result of subtracting two DATE values). Hence, the following call would generate all dates from the 1st of January 2006 to the 31st of January 2006.

```
DATE_RANGE('2006-01-01', '2006-01-31', 1)
```

Alternatively, the following call can be used to generate the 1st day of each month in the year 2006:

```
DATE_RANGE('2006-01-01', '2006-12-01', 100)
```

Note that 100 does *not* mean increment by 100 days each time, but by 1 month each time because the digit 1 falls in the MM part of YYYYMMDD. If **STEP** is omitted it defaults to 1 day.

Parameters

START The date (specified as a DATE, TIMESTAMP, or VARCHAR(26)) from which to start generating dates. **START** will always be part of the resulting table.

FINISH The date (specified as a DATE, TIMESTAMP, or VARCHAR(26)) on which to stop generating dates. **FINISH** may be part of the resulting table if iteration stops on **FINISH**. However, if the specified **STEP** causes iteration to overshoot **FINISH**, it will not be included.

STEP If provided, the duration by which to increment each row of the output. Specified as a date duration; a DECIMAL(8,0) value formatted as YYYYMMDD (numebr of years, number of months, number of days).

Returns

D The function returns a table with a single column simply named *D* which contains the dates generated.

Examples

Generate all days in the first month of 2010:

```
SELECT D
FROM TABLE(
  DATE_RANGE(MONTHSTART(2010, 1), MONTHEND(2010, 1))
);
```

```
D
----------
2010-01-01
2010-01-02
2010-01-03
2010-01-04
2010-01-05
```

```
2010-01-06
2010-01-07
2010-01-08
2010-01-09
2010-01-10
2010-01-11
2010-01-12
2010-01-13
2010-01-14
2010-01-15
2010-01-16
2010-01-17
2010-01-18
2010-01-19
2010-01-20
2010-01-21
2010-01-22
2010-01-23
2010-01-24
2010-01-25
2010-01-26
2010-01-27
2010-01-28
2010-01-29
2010-01-30
2010-01-31
```

Generate the first day of each month in 2010:

```
SELECT D
FROM TABLE(
  DATE_RANGE(YEARSTART(2010), YEAREND(2010), 100)
);
```

```
D
----------
2010-01-01
2010-02-01
2010-03-01
2010-04-01
2010-05-01
2010-06-01
2010-07-01
2010-08-01
2010-09-01
2010-10-01
2010-11-01
2010-12-01
```

Generate the last day of each month in 2010:

```
SELECT MONTHEND(D) AS D
FROM TABLE(
  DATE_RANGE(YEARSTART(2010), YEAREND(2010), 100)
);
```

```
D
----------
2010-01-31
2010-02-28
2010-03-31
2010-04-30
2010-05-31
2010-06-30
2010-07-31
2010-08-31
2010-09-30
2010-10-31
2010-11-30
2010-12-31
```

Calculate the number of days in each quarter of 2010 (this is a crude and inefficient method, but it serves to demonstrate the ability to aggregate result sets over date ranges):

```
SELECT QUARTER(D) AS Q, COUNT(*) AS DAYS
FROM TABLE(
  DATE_RANGE(YEARSTART(2010), YEAREND(2010))
)
GROUP BY QUARTER(D);
```

```
Q           DAYS
----------- -----------
          1          90
          2          91
          3          92
          4          92
```

See Also

- Source code
- *DATE scalar function*
- DATE (built-in function)
- DAYS (built-in function)

EXPORT_SCHEMA table function

Generates EXPORT commands for all tables in the specified schema, including or excluding generated and/or identity columns as requested.

Prototypes

```
EXPORT_SCHEMA(ASCHEMA VARCHAR(128), INCLUDE_GENERATED VARCHAR(1), INCLUDE_IDENTITY␣
↪VARCHAR(1))
EXPORT_SCHEMA(INCLUDE_GENERATED VARCHAR(1), INCLUDE_IDENTITY VARCHAR(1))
EXPORT_SCHEMA()

RETURNS TABLE(
    TABSCHEMA VARCHAR(128),
    TABNAME VARCHAR(128),
    SQL VARCHAR(8000)
)
```

Description

This table function can be used to generate a script containing EXPORT commands for all tables (not views) in the specified schema or the current schema if the **ASCHEMA** parameter is omitted. This is intended to be used in scripts for migrating databases or generating ETL scripts.

The generated EXPORT commands will target an *IXF* file named after the table, e.g. if **ASCHEMA** is *DATAMART*, and the table is *COUNTRIES* the file would be named `"DATAMART.COUNTRIES.IXF"`. The export command will explicitly name all columns in the table. Likewise, *LOAD_SCHEMA table function* generates LOAD commands with explicitly named columns. This is to ensure that if the target database's tables are not declared in exactly the same order as the source database, the transfer will still work if, for example, columns have been added to tables in the source but in the table declaration, they were not placed at the end of the table.

If the optional **INCLUDE_GENERATED** parameter is `'Y'` (the default), GENERATED ALWAYS columns will be included, otherwise they are excluded. GENERATED BY DEFAULT columns are always included. If the optional **INCLUDE_IDENTITY** parameter is `'Y'` (the default), IDENTITY columns will be included, otherwise they are excluded.

Parameters

ASCHEMA If provided, the schema containing the tables to generate EXPORT commands for. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

INCLUDE_GENERATED If this parameter is `'Y'` then any columns defined as GENERATED in the source tables will be included in the result. Contrariwise, if `'N'`, generated columns will be excluded. Defaults to `'Y'` if omitted.

INCLUDE_IDENTITY If this parameter is `'Y'` (and **INCLUDE_GENERATED** is `'Y'` given that identity columns are by definition generated) then any columns defined as IDENTITY in the source tables will be included in the result. Contrariwise, if `'N'`, identity columns will be excluded (regardless of the value of **INCLUDE_GENERATED**). Defaults to `'Y'` if omitted.

Returns

The function returns one row per table present in the source schema. Note that the function does *not* filter out invalidated or inoperative tables. The result table contains three columns:

TABSCHEMA Contains the name of the schema containing the table named in *TABNAME*.

TABNAME Contains the name of the table that will be exported by the command in the *SQL* column.

SQL Contains the text of the generated EXPORT command.

The purpose of including the (otherwise redundant) *TABSCHEMA* and *TABNAME* columns is to permit the result to be filtered further without having to dissect the *SQL* column.

Examples

Generated EXPORT commands for all tables in the current schema, excluding all generated columns:

```
SELECT SQL FROM TABLE(EXPORT_SCHEMA('N', 'N'))
```

```
SQL
---------------------------------------------------------------------------------------
↪---------------------------------------------------------------------------------------
↪--
EXPORT TO "DB2INST1.CL_SCHED.IXF" OF IXF SELECT CLASS_CODE,DAY,STARTING,ENDING FROM␣
↪DB2INST1.CL_SCHED
EXPORT TO "DB2INST1.DEPARTMENT.IXF" OF IXF SELECT DEPTNO,DEPTNAME,MGRNO,ADMRDEPT,
↪LOCATION FROM DB2INST1.DEPARTMENT
EXPORT TO "DB2INST1.ACT.IXF" OF IXF SELECT ACTNO,ACTKWD,ACTDESC FROM DB2INST1.ACT
EXPORT TO "DB2INST1.EMPLOYEE.IXF" OF IXF SELECT EMPNO,FIRSTNME,MIDINIT,LASTNAME,
↪WORKDEPT,PHONENO,HIREDATE,JOB,EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM FROM DB2INST1.
↪EMPLOYEE
EXPORT TO "DB2INST1.EMP_PHOTO.IXF" OF IXF SELECT EMPNO,PHOTO_FORMAT,PICTURE FROM␣
↪DB2INST1.EMP_PHOTO
EXPORT TO "DB2INST1.EMP_RESUME.IXF" OF IXF SELECT EMPNO,RESUME_FORMAT,RESUME FROM␣
↪DB2INST1.EMP_RESUME
EXPORT TO "DB2INST1.PROJECT.IXF" OF IXF SELECT PROJNO,PROJNAME,DEPTNO,RESPEMP,PRSTAFF,
↪PRSTDATE,PRENDATE,MAJPROJ FROM DB2INST1.PROJECT
EXPORT TO "DB2INST1.PROJACT.IXF" OF IXF SELECT PROJNO,ACTNO,ACSTAFF,ACSTDATE,ACENDATE␣
↪FROM DB2INST1.PROJACT
EXPORT TO "DB2INST1.EMPPROJACT.IXF" OF IXF SELECT EMPNO,PROJNO,ACTNO,EMPTIME,EMSTDATE,
↪EMENDATE FROM DB2INST1.EMPPROJACT
EXPORT TO "DB2INST1.IN_TRAY.IXF" OF IXF SELECT RECEIVED,SOURCE,SUBJECT,NOTE_TEXT FROM␣
↪DB2INST1.IN_TRAY
EXPORT TO "DB2INST1.ORG.IXF" OF IXF SELECT DEPTNUMB,DEPTNAME,MANAGER,DIVISION,
↪LOCATION FROM DB2INST1.ORG
EXPORT TO "DB2INST1.STAFF.IXF" OF IXF SELECT ID,NAME,DEPT,JOB,YEARS,SALARY,COMM FROM␣
↪DB2INST1.STAFF
EXPORT TO "DB2INST1.SALES.IXF" OF IXF SELECT SALES_DATE,SALES_PERSON,REGION,SALES␣
↪FROM DB2INST1.SALES
EXPORT TO "DB2INST1.STAFFG.IXF" OF IXF SELECT ID,NAME,DEPT,JOB,YEARS,SALARY,COMM FROM␣
↪DB2INST1.STAFFG
EXPORT TO "DB2INST1.EMPMDC.IXF" OF IXF SELECT EMPNO,DEPT,DIV FROM DB2INST1.EMPMDC
EXPORT TO "DB2INST1.PRODUCT.IXF" OF IXF SELECT PID,NAME,PRICE,PROMOPRICE,PROMOSTART,
↪PROMOEND,DESCRIPTION FROM DB2INST1.PRODUCT
EXPORT TO "DB2INST1.INVENTORY.IXF" OF IXF SELECT PID,QUANTITY,LOCATION FROM DB2INST1.
↪INVENTORY
EXPORT TO "DB2INST1.CUSTOMER.IXF" OF IXF SELECT CID,INFO,HISTORY FROM DB2INST1.
↪CUSTOMER
EXPORT TO "DB2INST1.PURCHASEORDER.IXF" OF IXF SELECT POID,STATUS,CUSTID,ORDERDATE,
↪PORDER,COMMENTS FROM DB2INST1.PURCHASEORDER
EXPORT TO "DB2INST1.CATALOG.IXF" OF IXF SELECT NAME,CATLOG FROM DB2INST1.CATALOG
EXPORT TO "DB2INST1.SUPPLIERS.IXF" OF IXF SELECT SID,ADDR FROM DB2INST1.SUPPLIERS
EXPORT TO "DB2INST1.PRODUCTSUPPLIER.IXF" OF IXF SELECT PID,SID FROM DB2INST1.
↪PRODUCTSUPPLIER
```

Generate EXPORT commands for all tables in the DB2INST1 schema whose names begin with `'EMP'`, including generated columns which aren't also identity columns:

```
SELECT SQL
FROM TABLE(EXPORT_SCHEMA('DB2INST1', 'Y', 'N'))
WHERE TABNAME LIKE 'EMP%'
```

```
SQL
--------------------------------------------------------------------------------------
↪--------------------------------------------------------------------------------------
↪--
EXPORT TO "DB2INST1.EMPLOYEE.IXF" OF IXF SELECT EMPNO,FIRSTNME,MIDINIT,LASTNAME,
↪WORKDEPT,PHONENO,HIREDATE,JOB,EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM FROM DB2INST1.
↪EMPLOYEE
EXPORT TO "DB2INST1.EMPMDC.IXF" OF IXF SELECT EMPNO,DEPT,DIV FROM DB2INST1.EMPMDC
EXPORT TO "DB2INST1.EMPPROJACT.IXF" OF IXF SELECT EMPNO,PROJNO,ACTNO,EMPTIME,EMSTDATE,
↪EMENDATE FROM DB2INST1.EMPPROJACT
EXPORT TO "DB2INST1.EMP_PHOTO.IXF" OF IXF SELECT EMPNO,PHOTO_FORMAT,PICTURE FROM␣
↪DB2INST1.EMP_PHOTO
EXPORT TO "DB2INST1.EMP_RESUME.IXF" OF IXF SELECT EMPNO,RESUME_FORMAT,RESUME FROM␣
↪DB2INST1.EMP_RESUME
```

See Also

- Source code
- *EXPORT_TABLE scalar function*
- *LOAD_TABLE scalar function*
- *LOAD_SCHEMA table function*
- LOAD (built-in command)
- EXPORT (build-in command)

EXPORT_TABLE scalar function

Generates an EXPORT command for the specified table including or excluding generated and/or identity columns as requested.

Prototypes

```
EXPORT_TABLE(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1),␣
↪INCLUDE_IDENTITY VARCHAR(1))
EXPORT_TABLE(ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1), INCLUDE_IDENTITY␣
↪VARCHAR(1))
EXPORT_TABLE(ATABLE VARCHAR(128))

RETURNS VARCHAR(8000)
```

Description

This function generates an EXPORT command for the specified table in the specified schema or the current schema if **ASCHEMA** is omitted. If the optional **INCLUDE_GENERATED** parameter is `'Y'` (the default), GENERATED ALWAYS columns will be included, otherwise they are excluded. GENERATED BY DEFAULT columns are always included. If the optional **INCLUDE_IDENTITY** parameter is `'Y'` (the default), IDENTITY columns will be included, otherwise they are excluded.

See *EXPORT_SCHEMA table function* for more information on the generated command.

Parameters

ASCHEMA If provided, the schema containing the table to generate an EXPORT command for. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATABLE The name of the table to generate an EXPORT command for.

INCLUDE_GENERATED If this parameter is `'Y'` then any columns defined as GENERATED in the source table will be included in the export. Contrariwise, if `'N'`, generated columns will be excluded from the command. Defaults to `'Y'` if omitted.

INCLUDE_IDENTITY If this parameter is `'Y'` (and **INCLUDE_GENERATED** is `'Y'` given that identity columns are by definition generated) then any columns defined as IDENTITY in the source table will be included in the export. Contrariwise, if `'N'`, identity columns will be excluded from the command (regardless of the value of **INCLUDE_GENERATED**). Defaults to `'Y'` if omitted.

Examples

Generate an EXPORT command for the *EMPLOYEE* table in the standard *SAMPLE* database:

```
VALUES EXPORT_TABLE('EMPLOYEE')
```

```
1
--------------------------------------------------------------------------------
↪--------------------------------------------------------------------------------
↪--...
EXPORT TO "DB2INST1.EMPLOYEE.IXF" OF IXF SELECT EMPNO,FIRSTNME,MIDINIT,LASTNAME,
↪WORKDEPT,PHONENO,HIREDATE,JOB,EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM FROM DB2INST1.
↪EMPLOYEE
```

Generate an EXPORT command for the *PEOPLE* table (DDL included) excluding IDENTITY columns:

```
CREATE TABLE PEOPLE (
    ID      INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
    NAME    VARCHAR(100) DEFAULT '' NOT NULL,
    GENDER  CHAR(1) NOT NULL,
    DOB     DATE NOT NULL,
    TITLE   VARCHAR(10) NOT NULL GENERATED ALWAYS AS (
        CASE GENDER
            WHEN 'M' THEN 'Mr.'
            WHEN 'F' THEN 'Ms.'
        END
    ),
    CONSTRAINT GENDER_CK CHECK (GENDER IN ('M', 'F'))
);

VALUES EXPORT_TABLE('PEOPLE', 'Y', 'N');
```

```
1
--------------------------------------------------------------------------------
↪--...
EXPORT TO "DB2INST1.PEOPLE.IXF" OF IXF SELECT NAME,GENDER,DOB,TITLE FROM DB2INST1.
↪PEOPLE
```

See Also

- Source code
- *EXPORT_SCHEMA table function*
- *LOAD_TABLE scalar function*
- *LOAD_SCHEMA table function*
- LOAD (built-in command)
- EXPORT (build-in command)

HOUREND scalar function

Returns a TIMESTAMP at the end of **AHOUR** on the date **AYEAR**, **AMONTH**, **ADAY**, or at the end of the hour of **ATIMESTAMP**.

Prototypes

```
HOUREND(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER, AHOUR INTEGER)
HOUREND(ATIMESTAMP TIMESTAMP)
HOUREND(ATIMESTAMP VARCHAR(26))

RETURNS TIMESTAMP
```

Description

Returns a TIMESTAMP value representing the last microsecond of **AHOUR** in the date given by **AYEAR**, **AMONTH**, and **ADAY**, or of the timestamp given by **ATIMESTAMP** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year component of the resulting timestamp.

AMONTH If provided, the month component of the resulting timestamp.

ADAY If provided, the day component of the resulting timestamp.

AHOUR If provided, the hour component of the resulting timestamp.

ATIMESTAMP If provided, the timestamp from which to derive the end of the hour. Either **AYEAR**, **AMONTH**, **ADAY**, and **AHOUR**, or **ATIMESTAMP** must be provided.

Examples

Calculate the last microsecond of the specified hour:

```
VALUES HOUREND('2010-01-23 04:56:00');
```

```
1
--------------------------
2010-01-23-04.59.59.999999
```

Calculate the end of the first working day in 2011:

```
VALUES HOUREND(2011, 1, DAY(
  CASE WHEN DAYOFWEEK(YEARSTART(2011)) IN (1, 7)
    THEN NEXT_DAYOFWEEK(YEARSTART(2011), 2)
    ELSE YEARSTART(2011)
  END), 4);
```

```
1
--------------------------
2011-01-03-04.59.59.999999
```

See Also

- Source code
- *HOURSTART scalar function*
- HOUR (built-in function)

HOURSTART scalar function

Returns a TIMESTAMP at the start of **AHOUR** on the date **AYEAR**, **AMONTH**, **ADAY**, or at the start of the hour of **ATIMESTAMP**.

Prototypes

```
HOURSTART(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER, AHOUR INTEGER)
HOURSTART(ATIMESTAMP TIMESTAMP)
HOURSTART(ATIMESTAMP VARCHAR(26))

RETURNS TIMESTAMP
```

Description

Returns a TIMESTAMP value representing the first microsecond of **AHOUR** in the date given by **AYEAR**, **AMONTH**, and **ADAY**, or of the timestamp given by **ATIMESTAMP** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year component of the resulting timestamp.

AMONTH If provided, the month component of the resulting timestamp.

ADAY If provided, the day component of the resulting timestamp.

AHOUR If provided, the hour component of the resulting timestamp.

ATIMESTAMP If provided, the timestamp from which to derive the start of the hour. Either **AYEAR**, **AMONTH**, **ADAY**, and **AHOUR**, or **ATIMESTAMP** must be provided.

Examples

Truncate the specified timestamp to the nearest hour:

```
VALUES HOURSTART('2010-01-23 04:56:00');
```

```
1
--------------------------
2010-01-23-04.00.00.000000
```

Calculate the start of the first working day in 2011:

```
VALUES HOURSTART(2011, 1, DAY(
  CASE WHEN DAYOFWEEK(YEARSTART(2011)) IN (1, 7)
    THEN NEXT_DAYOFWEEK(YEARSTART(2011), 2)
    ELSE YEARSTART(2011)
  END), 9);
```

```
1
--------------------------
2011-01-03-09.00.00.000000
```

See Also

- Source code
- *HOUREND scalar function*
- HOUR (built-in function)

LOAD_SCHEMA table function

Generates LOAD commands for all tables in the specified schema, ignoring or overriding generated and/or identity columns as requested.

Prototypes

```
LOAD_SCHEMA(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1),␣
↪INCLUDE_IDENTITY VARCHAR(1))
LOAD_SCHEMA(ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1), INCLUDE_IDENTITY␣
↪VARCHAR(1))
LOAD_SCHEMA(ATABLE VARCHAR(128))

RETURNS TABLE(
    TABSCHEMA VARCHAR(128),
    TABNAME VARCHAR(128),
    SQL VARCHAR(8000)
)
```

Description

This table function can be used to generate a script containing LOAD commands for all tables (not views) in the specified schema or the current schema if the **ASCHEMA** parameter is omitted. This is intended to be used in scripts for migrating the database.

This function is the counterpart of *EXPORT_SCHEMA table function*. See *EXPORT_SCHEMA table function* and *LOAD_TABLE scalar function* function for more information on the commands generated.

Parameters

ASCHEMA If provided, the schema containing the tables to generate LOAD commands for. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

INCLUDE_GENERATED If this parameter is `'Y'` then the routine assumes generated columns are included in the source files, and the LOAD commands will include the *GENERATEDOVERRIDE* modifier. Otherwise, if `'N'`, the *GENERATEDMISSING* modifier will be used instead. Defaults to `'Y'` if omitted.

INCLUDE_IDENTITY If this parameter is `'Y'` then the routine assumes identity columns are included in the source files, and the LOAD commands will include the *IDENTITYOVERRIDE* modifier. Otherwise, if `'N'`, the *IDENTITYMISSING* modifier will be used instead. Defaults to `'Y'` if omitted.

Returns

The function returns one row per table present in the source schema. Note that the function does *not* filter out invalidated or inoperative tables. The result table contains three columns:

TABSCHEMA Contains the name of the schema containing the table named in *TABNAME*.

TABNAME Contains the name of the table that will be loaded by the command in the *SQL* column.

SQL Contains the text of the generated LOAD command.

The purpose of including the (otherwise redundant) *TABSCHEMA* and *TABNAME* columns is to permit the result to be filtered further without having to dissect the *SQL* column.

Examples

Generated LOAD commands for all tables in the current schema, excluding all generated columns:

```
SELECT SQL FROM TABLE(LOAD_SCHEMA('N', 'N'))
```

```
SQL
------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
↪----------------------------------
LOAD FROM "DB2INST1.CL_SCHED.IXF" OF IXF METHOD N (CLASS_CODE,DAY,STARTING,ENDING)␣
↪REPLACE INTO DB2INST1.CL_SCHED (CLASS_CODE,DAY,STARTING,ENDING)
LOAD FROM "DB2INST1.DEPARTMENT.IXF" OF IXF METHOD N (DEPTNO,DEPTNAME,MGRNO,ADMRDEPT,
↪LOCATION) REPLACE INTO DB2INST1.DEPARTMENT (DEPTNO,DEPTNAME,MGRNO,ADMRDEPT,LOCATION)
LOAD FROM "DB2INST1.ACT.IXF" OF IXF METHOD N (ACTNO,ACTKWD,ACTDESC) REPLACE INTO␣
↪DB2INST1.ACT (ACTNO,ACTKWD,ACTDESC)
LOAD FROM "DB2INST1.EMPLOYEE.IXF" OF IXF METHOD N (EMPNO,FIRSTNME,MIDINIT,LASTNAME,
↪WORKDEPT,PHONENO,HIREDATE,JOB,EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM) REPLACE INTO␣
↪DB2INST1.EMPLOYEE (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,PHONENO,HIREDATE,JOB,
↪EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM)
```

```
LOAD FROM "DB2INST1.EMP_PHOTO.IXF" OF IXF METHOD N (EMPNO,PHOTO_FORMAT,PICTURE)␣
↪REPLACE INTO DB2INST1.EMP_PHOTO (EMPNO,PHOTO_FORMAT,PICTURE)
LOAD FROM "DB2INST1.EMP_RESUME.IXF" OF IXF METHOD N (EMPNO,RESUME_FORMAT,RESUME)␣
↪REPLACE INTO DB2INST1.EMP_RESUME (EMPNO,RESUME_FORMAT,RESUME)
LOAD FROM "DB2INST1.PROJECT.IXF" OF IXF METHOD N (PROJNO,PROJNAME,DEPTNO,RESPEMP,
↪PRSTAFF,PRSTDATE,PRENDATE,MAJPROJ) REPLACE INTO DB2INST1.PROJECT (PROJNO,PROJNAME,
↪DEPTNO,RESPEMP,PRSTAFF,PRSTDATE,PRENDATE,MAJPROJ)
LOAD FROM "DB2INST1.PROJACT.IXF" OF IXF METHOD N (PROJNO,ACTNO,ACSTAFF,ACSTDATE,
↪ACENDATE) REPLACE INTO DB2INST1.PROJACT (PROJNO,ACTNO,ACSTAFF,ACSTDATE,ACENDATE)
LOAD FROM "DB2INST1.EMPPROJACT.IXF" OF IXF METHOD N (EMPNO,PROJNO,ACTNO,EMPTIME,
↪EMSTDATE,EMENDATE) REPLACE INTO DB2INST1.EMPPROJACT (EMPNO,PROJNO,ACTNO,EMPTIME,
↪EMSTDATE,EMENDATE)
LOAD FROM "DB2INST1.IN_TRAY.IXF" OF IXF METHOD N (RECEIVED,SOURCE,SUBJECT,NOTE_TEXT)␣
↪REPLACE INTO DB2INST1.IN_TRAY (RECEIVED,SOURCE,SUBJECT,NOTE_TEXT)
LOAD FROM "DB2INST1.ORG.IXF" OF IXF METHOD N (DEPTNUMB,DEPTNAME,MANAGER,DIVISION,
↪LOCATION) REPLACE INTO DB2INST1.ORG (DEPTNUMB,DEPTNAME,MANAGER,DIVISION,LOCATION)
LOAD FROM "DB2INST1.STAFF.IXF" OF IXF METHOD N (ID,NAME,DEPT,JOB,YEARS,SALARY,COMM)␣
↪REPLACE INTO DB2INST1.STAFF (ID,NAME,DEPT,JOB,YEARS,SALARY,COMM)
LOAD FROM "DB2INST1.SALES.IXF" OF IXF METHOD N (SALES_DATE,SALES_PERSON,REGION,SALES)␣
↪REPLACE INTO DB2INST1.SALES (SALES_DATE,SALES_PERSON,REGION,SALES)
LOAD FROM "DB2INST1.STAFFG.IXF" OF IXF METHOD N (ID,NAME,DEPT,JOB,YEARS,SALARY,COMM)␣
↪REPLACE INTO DB2INST1.STAFFG (ID,NAME,DEPT,JOB,YEARS,SALARY,COMM)
LOAD FROM "DB2INST1.EMPMDC.IXF" OF IXF METHOD N (EMPNO,DEPT,DIV) REPLACE INTO␣
↪DB2INST1.EMPMDC (EMPNO,DEPT,DIV)
LOAD FROM "DB2INST1.PRODUCT.IXF" OF IXF METHOD N (PID,NAME,PRICE,PROMOPRICE,
↪PROMOSTART,PROMOEND,DESCRIPTION) REPLACE INTO DB2INST1.PRODUCT (PID,NAME,PRICE,
↪PROMOPRICE,PROMOSTART,PROMOEND,DESCRIPTION)
LOAD FROM "DB2INST1.INVENTORY.IXF" OF IXF METHOD N (PID,QUANTITY,LOCATION) REPLACE␣
↪INTO DB2INST1.INVENTORY (PID,QUANTITY,LOCATION)
LOAD FROM "DB2INST1.CUSTOMER.IXF" OF IXF METHOD N (CID,INFO,HISTORY) REPLACE INTO␣
↪DB2INST1.CUSTOMER (CID,INFO,HISTORY)
LOAD FROM "DB2INST1.PURCHASEORDER.IXF" OF IXF METHOD N (POID,STATUS,CUSTID,ORDERDATE,
↪PORDER,COMMENTS) REPLACE INTO DB2INST1.PURCHASEORDER (POID,STATUS,CUSTID,ORDERDATE,
↪PORDER,COMMENTS)
LOAD FROM "DB2INST1.CATALOG.IXF" OF IXF METHOD N (NAME,CATLOG) REPLACE INTO DB2INST1.
↪CATALOG (NAME,CATLOG)
LOAD FROM "DB2INST1.SUPPLIERS.IXF" OF IXF METHOD N (SID,ADDR) REPLACE INTO DB2INST1.
↪SUPPLIERS (SID,ADDR)
LOAD FROM "DB2INST1.PRODUCTSUPPLIER.IXF" OF IXF METHOD N (PID,SID) REPLACE INTO␣
↪DB2INST1.PRODUCTSUPPLIER (PID,SID)
```

Generate LOAD commands for all tables in the DB2INST1 schema whose names begin with `'EMP'`, including generated columns which aren't also identity columns:

```
SELECT SQL
FROM TABLE(LOAD_SCHEMA('DB2INST1', 'Y', 'N'))
WHERE TABNAME LIKE 'EMP%'
```

```
SQL
-------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
↪----------------------------------
LOAD FROM "DB2INST1.EMPLOYEE.IXF" OF IXF METHOD N (EMPNO,FIRSTNME,MIDINIT,LASTNAME,
↪WORKDEPT,PHONENO,HIREDATE,JOB,EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM) REPLACE INTO␣
↪DB2INST1.EMPLOYEE (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,PHONENO,HIREDATE,JOB,
↪EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM)
```

```
LOAD FROM "DB2INST1.EMPMDC.IXF" OF IXF METHOD N (EMPNO,DEPT,DIV) REPLACE INTO␣
↪DB2INST1.EMPMDC (EMPNO,DEPT,DIV)
LOAD FROM "DB2INST1.EMPPROJACT.IXF" OF IXF METHOD N (EMPNO,PROJNO,ACTNO,EMPTIME,
↪EMSTDATE,EMENDATE) REPLACE INTO DB2INST1.EMPPROJACT (EMPNO,PROJNO,ACTNO,EMPTIME,
↪EMSTDATE,EMENDATE)
LOAD FROM "DB2INST1.EMP_PHOTO.IXF" OF IXF METHOD N (EMPNO,PHOTO_FORMAT,PICTURE)␣
↪REPLACE INTO DB2INST1.EMP_PHOTO (EMPNO,PHOTO_FORMAT,PICTURE)
LOAD FROM "DB2INST1.EMP_RESUME.IXF" OF IXF METHOD N (EMPNO,RESUME_FORMAT,RESUME)␣
↪REPLACE INTO DB2INST1.EMP_RESUME (EMPNO,RESUME_FORMAT,RESUME)
```

See Also

- Source code
- *EXPORT_TABLE scalar function*
- *EXPORT_SCHEMA table function*
- *LOAD_TABLE scalar function*
- LOAD (built-in command)
- EXPORT (built-in command)

LOAD_TABLE scalar function

Generates a LOAD command for the specified table including or excluding generated and/or identity columns as requested.

Prototypes

```
LOAD_TABLE(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1),␣
↪INCLUDE_IDENTITY VARCHAR(1))
LOAD_TABLE(ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1), INCLUDE_IDENTITY␣
↪VARCHAR(1))
LOAD_TABLE(ATABLE VARCHAR(128))

RETURNS VARCHAR(8000)
```

Description

This function generates a LOAD command for the specified table in the specified schema or the current schema if **ASCHEMA** is omitted. If the optional **INCLUDE_GENERATED** parameter is `'Y'` (the default), GENERATED ALWAYS columns are assumed to be included in the source file, and the LOAD command will utilize *GENERATEDOVERRIDE*, otherwise the LOAD command will utilize *GENERATEDMISSING*. GENERATED BY DEFAULT columns are treated as ordinary columns. If the optional **INCLUDE_IDENTITY** parameter is `'Y'` (the default), IDENTITY columns are assumed to be included in the source file, and the LOAD command will utilize *IDENTITYOVERRIDE*, otherwise the LOAD command will utilize *IDENTITYMISSING*.

See *EXPORT_SCHEMA table function* for more information on the generated command.

Parameters

ASCHEMA If provided, the schema containing the table to generate a LOAD command for. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATABLE The name of the table to generate a LOAD command for.

INCLUDE_GENERATED If this parameter is `'Y'` then the routine assumes generated columns are included in the source file, and the LOAD command will include the *GENERATEDOVERRIDE* modifier. Otherwise, if `'N'`, the *GENERATEDMISSING* modifier will be used instead. Defaults to `'Y'` if omitted.

INCLUDE_IDENTITY If this parameter is `'Y'` then the routine assumes identity columns are included in the source file, and the LOAD command will include the *IDENTITYOVERRIDE* modifier. Otherwise, if `'N'`, the *IDENTITYMISSING* modifier will be used instead. Defaults to `'Y'` if omitted.

Examples

Generate a LOAD command for the *EMPLOYEE* table in the standard *SAMPLE* database:

```
VALUES LOAD_TABLE('EMPLOYEE')
```

```
1
------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
↪--...
LOAD FROM "DB2INST1.EMPLOYEE.IXF" OF IXF METHOD N (EMPNO,FIRSTNME,MIDINIT,LASTNAME,
↪WORKDEPT,PHONENO,HIREDATE,JOB,EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM) REPLACE INTO␣
↪DB2INST1.EMPLOYEE (EMPNO,FIRSTNME,MIDINIT,LASTNAME,WORKDEPT,PHONENO,HIREDATE,JOB,
↪EDLEVEL,SEX,BIRTHDATE,SALARY,BONUS,COMM)
```

Generate a LOAD command for the *PEOPLE* table (DDL included) excluding IDENTITY columns:

```
CREATE TABLE PEOPLE (
    ID      INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY,
    NAME    VARCHAR(100) DEFAULT '' NOT NULL,
    GENDER  CHAR(1) NOT NULL,
    DOB     DATE NOT NULL,
    TITLE   VARCHAR(10) NOT NULL GENERATED ALWAYS AS (
        CASE GENDER
            WHEN 'M' THEN 'Mr.'
            WHEN 'F' THEN 'Ms.'
        END
    ),
    CONSTRAINT GENDER_CK CHECK (GENDER IN ('M', 'F'))
);

VALUES LOAD_TABLE('PEOPLE', 'Y', 'N');
```

```
1
------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------
LOAD FROM "DB2INST1.PEOPLE.IXF" OF IXF MODIFIED BY GENERATEDOVERRIDE,IDENTITYMISSING␣
↪METHOD N (NAME,GENDER,DOB,TITLE) REPLACE INTO DB2INST1.PEOPLE (NAME,GENDER,DOB,
↪TITLE)
```

See Also

- Source code
- *EXPORT_TABLE scalar function*
- *EXPORT_SCHEMA table function*
- *LOAD_SCHEMA table function*
- LOAD (built-in command)
- EXPORT (build-in command)

MINUTEEND scalar function

Returns a TIMESTAMP at the end of **AHOUR:AMINUTE** on the date **AYEAR**, **AMONTH**, **ADAY**, or at the end of the minute of **ATIMESTAMP**.

Prototypes

```
MINUTEEND(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER, AHOUR INTEGER, AMINUTE INTEGER)
MINUTEEND(ATIMESTAMP TIMESTAMP)
MINUTEEND(ATIMESTAMP VARCHAR(26))

RETURNS TIMESTAMP
```

Description

Returns a TIMESTAMP value representing the last microsecond of **AMINUTE** in **AHOUR** on the date given by **AYEAR**, **AMONTH**, and **ADAY**, or of the timestamp given by **ATIMESTAMP** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year component of the resulting timestamp.

AMONTH If provided, the month component of the resulting timestamp.

ADAY If provided, the day component of the resulting timestamp.

AHOUR If provided, the hour component of the resulting timestamp.

AMINUTE If provided, the minute component of the resulting timestamp.

ATIMESTAMP If provided, the timestamp from which to derive the end of the minute. Either **AYEAR**, **AMONTH**, **ADAY**, **AHOUR**, and **AMINUTE**, or **ATIMESTAMP** must be provided.

Examples

Round the specified timestamp up to one microsecond before the next minute:

```
VALUES MINUTEEND('2010-01-23 04:56:12');
```

```
1
--------------------------
2010-01-23-04.56.59.999999
```

Generate a timestamp at the end of a minute with the specified fields:

```
VALUES MINUTEEND(2010, 2, 14, 9, 30);
```

```
1
--------------------------
2010-02-14-09.30.59.999999
```

See Also

- Source code
- *MINUTESTART scalar function*
- MINUTE (built-in function)

MINUTESTART scalar function

Returns a TIMESTAMP at the start of **AHOUR:AMINUTE** on the date **AYEAR**, **AMONTH**, **ADAY**, or at the start of the minute of **ATIMESTAMP**.

Prototypes

```
MINUTESTART(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER, AHOUR INTEGER, AMINUTE␣
↪INTEGER)
MINUTESTART(ATIMESTAMP TIMESTAMP)
MINUTESTART(ATIMESTAMP VARCHAR(26))

RETURNS TIMESTAMP
```

Description

Returns a TIMESTAMP value representing the first microsecond of **AMINUTE** in **AHOUR** on the date given by **AYEAR**, **AMONTH**, and **ADAY**, or of the timestamp given by **ATIMESTAMP** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year component of the resulting timestamp.

AMONTH If provided, the month component of the resulting timestamp.

ADAY If provided, the day component of the resulting timestamp.

AHOUR If provided, the hour component of the resulting timestamp.

AMINUTE If provided, the minute component of the resulting timestamp.

ATIMESTAMP If provided, the timestamp from which to derive the start of the minute. Either **AYEAR**, **AMONTH**, **ADAY**, **AHOUR**, and **AMINUTE**, or **ATIMESTAMP** must be provided.

Examples

Truncate the specified timestamp to the nearest minute:

```
VALUES MINUTESTART('2010-01-23 04:56:12');
```

```
1
--------------------------
2010-01-23-04.56.00.000000
```

Generate a timestamp at the start of a minute with the specified fields:

```
VALUES MINUTESTART(2010, 2, 14, 9, 30);
```

```
1
--------------------------
2010-02-14-09.30.00.000000
```

See Also

- Source code
- *MINUTEEND scalar function*
- MINUTE (built-in function)

MONTHEND scalar function

Returns the last day of month **AMONTH** in the year **AYEAR**, or the last day of the month of **ADATE**.

Prototypes

```
MONTHEND(AYEAR INTEGER, AMONTH INTEGER)
MONTHEND(ADATE DATE)
MONTHEND(ADATE TIMESTAMP)
MONTHEND(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the last day of **AMONTH** in **AYEAR**, or the last day of the month of **ADATE** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AMONTH** for which to return the ending date.

AMONTH If provided, the month for which to return to the ending date.

ADATE If provided the date in the month for which to return the ending date. Either **AYEAR** and **AMONTH**, or **ADATE** must be specified.

Examples

Calculate the ending date of the second month of 2010:

```
VALUES MONTHEND(2010, 2);
```

```
1
----------
2010-02-28
```

Calculate the ending date for the 28th of January, 2009:

```
VALUES MONTHEND('2009-01-28');
```

```
1
----------
2009-01-31
```

See Also

- Source code
- *MONTHSTART scalar function*
- MONTH (built-in function)

MONTHSTART scalar function

Returns the first day of the month that **ADATE** exists within, or the first day of the month **AMONTH** in the year **AYEAR**.

Prototypes

```
MONTHSTART(AYEAR INTEGER, AMONTH INTEGER)
MONTHSTART(ADATE DATE)
MONTHSTART(ADATE TIMESTAMP)
MONTHSTART(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the first day of **AMONTH** in **AYEAR**, or the first day of the month of **ADATE** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AMONTH** for which to return the starting date.

AMONTH If provided, the month for which to return to the starting date.

ADATE If provided the date in the month for which to return the starting date. Either **AYEAR** and **AMONTH**, or **ADATE** must be specified.

Examples

Calculate the starting date of the second month in 2010:

```
VALUES MONTHSTART(2010, 2);
```

```
1
----------
2010-02-01
```

Calculate the start of the month for the 28th of January, 2009:

```
VALUES MONTHSTART('2009-01-28');
```

```
1
----------
2009-01-01
```

See Also

- Source code
- *MONTHEND scalar function*
- MONTH (built-in function)

MONTHWEEK scalar function

Returns the week of the month that **ADATE** exists within (weeks start on a Sunday, result will be in the range 1-6).

Prototypes

```
MONTHWEEK(ADATE DATE)
MONTHWEEK(ADATE TIMESTAMP)
MONTHWEEK(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the week of the month of **ADATE**, where weeks start on a Sunday. The result will be in the range 1-6 as partial weeks are permitted. For example, if the first day of a month is a Saturday, it will be counted as week 1, which lasts one day. The next day, Sunday, will start week 2.

Parameters

ADATE The date to calculate the week of the month for.

Examples

Calculate the week of the month for the 31st of January, 2010:

```
VALUES MONTHWEEK(DATE(2010, 1, 31));
```

```
1
------
     6
```

Calculate the length of all weeks in January 2010:

```
SELECT MONTHWEEK(D) AS WEEK_NUM, COUNT(*) AS WEEK_LENGTH
FROM TABLE(DATE_RANGE(MONTHSTART(2010, 1), MONTHEND(2010, 1)))
GROUP BY MONTHWEEK(D);
```

```
WEEK_NUM WEEK_LENGTH
-------- -----------
       1           2
       2           7
       3           7
       4           7
       5           7
       6           1
```

See Also

- Source code
- *MONTHWEEK_ISO scalar function*

MONTHWEEK_ISO scalar function

Returns the week of the month that **ADATE** exists within (weeks start on a Monday, result will be in the range 1-6).

Prototypes

```
MONTHWEEK_ISO(ADATE DATE)
MONTHWEEK_ISO(ADATE TIMESTAMP)
MONTHWEEK_ISO(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the week of the month of **ADATE**, where weeks start on a Monday. The result will be in the range 1-6 as partial weeks are permitted. For example, if the first day of a month is a Sunday, it will be counted as week 1, which lasts one day. The next day, Monday, will start week 2.

Parameters

ADATE The date to calculate the week of the month for.

Examples

Calculate the week of the month for the 31st of January, 2010:

```
VALUES MONTHWEEK(DATE(2010, 1, 31));
```

```
1
------
     5
```

Calculate the length of all weeks in January 2010:

```
SELECT MONTHWEEK_ISO(D) AS WEEK_NUM, COUNT(*) AS WEEK_LENGTH
FROM TABLE(DATE_RANGE(MONTHSTART(2010, 1), MONTHEND(2010, 1)))
GROUP BY MONTHWEEK_ISO(D);
```

```
WEEK_NUM WEEK_LENGTH
-------- -----------
       1           3
       2           7
       3           7
       4           7
       5           7
```

See Also

- Source code
- *MONTHWEEK scalar function*

NEXT_DAYOFWEEK scalar function

Returns the earliest date later than **ADATE**, which is also a particular day of the week, **ADOW** (1=Sunday, 2=Monday, 6=Saturday, etc.)

Prototypes

```
NEXT_DAYOFWEEK(ADATE DATE, ADOW INTEGER)
NEXT_DAYOFWEEK(ADATE TIMESTAMP, ADOW INTEGER)
NEXY_DAYOFWEEK(ADATE VARCHAR(26), ADOW INTEGER)
NEXT_DAYOFWEEK(ADOW INTEGER)

RETURNS DATE
```

Description

Returns the specified day of the week following the given date. Days of the week are specified in the same fashion as the built-in *DAYOFWEEK* function (i.e. 1=Sunday, 2=Monday, … 7=Saturday). If **ADATE** is omitted the current date is used.

Parameters

ADATE The date after which to return a specific day of the week. If this parameter is omitted the *CURRENT DATE* special register is used.

ADOW The day of the week to find specified as an integer where 1 represents Sunday, 2 is Monday, and so on.

Examples

Find the next Monday after the start of 2010:

```
VALUES VARCHAR(NEXT_DAYOFWEEK(YEARSTART(2010), 2), ISO);
```

```
1
----------
2010-01-04
```

Find the third Thursday in February 2010 (note, the CASE expression is necessary in case February starts on a Thursday, in which case *NEXT_DAYOFWEEK* will be returning the date of the second Thursday in the month, not the first):

```
VALUES VARCHAR(NEXT_DAYOFWEEK(MONTHSTART(2010, 2), 5) +
  CASE DAYOFWEEK(MONTHSTART(2010, 2))
    WHEN 5 THEN 7
    ELSE 14
  END DAYS, ISO);
```

```
1
----------
2010-02-18
```

See Also

- Source code
- *PRIOR_DAYOFWEEK scalar function*

PCRE_GROUPS table function

Searches for regular expression **PATTERN** in **TEXT**, returning a table detailing all matched groups.

Prototypes

```
PCRE_GROUPS(PATTERN VARCHAR(1000), TEXT VARCHAR(4000))

RETURNS TABLE(
  GROUP INTEGER,
  POSITION INTEGER,
  CONTENT VARCHAR(4000)
)
```

Description

PCRE groups table function. Given a regular expression in **PATTERN**, and some text to search in **TEXT**, the function performs a search for **PATTERN** in the text and returns the result as a table containing a row for each matching group (including group 0 which implicitly covers the entire search pattern).

Parameters

PATTERN The Perl-compatible Regular Expression (PCRE) to search for.

TEXT The text to search within.

Returns

GROUP The index of the capturing group; group 0 represents the portion of **TEXT** which matched the entire **PATTERN**.

POSITION The 1-based position of the group within **TEXT**.

CONTENT The content of the matched group.

Examples

This example demonstrates how multiple groups are matched and returned by the function:

```
SELECT
    T.GROUP,
    T.POSITION,
    T.CONTENT
FROM
    TABLE(
        PCRE_GROUPS('(<([A-Z][A-Z0-9]*)[^>]*>)(.*?)(</\2>)', '<B>BOLD!</B>')
    ) AS T
```

```
GROUP  POSITION  CONTENT
-----  --------  -------------------------
    0         1  <B>BOLD!</B>
    1         1  <B>
    2         2  B
    3         4  BOLD!
    4         9  </B>
```

Example demonstrating how unmatched groups are not returned, while groups matching the empty string are:

```
SELECT
    T.GROUP,
    T.POSITION,
    T.CONTENT
FROM
    TABLE(
        PCRE_GROUPS('(FOO)?(\s?)(BAR)?(\s?)(BAZ)?', 'FOOBAR')
    ) AS T
```

```
GROUP  POSITION  CONTENT
-----  --------  -------------------------
    0         1  FOOBAR
    1         1  FOO
    2         4
    3         4  BAR
    4         7
```

See Also

- SQL source code
- C source code
- *PCRE_SEARCH scalar function*
- *PCRE_SUB scalar function*
- *PCRE_SPLIT table function*
- PCRE library homepage
- Wikipedia PCRE article

PCRE_SEARCH scalar function

Searches for regular expression **PATTERN** within **TEXT** starting at 1-based **START**.

Prototypes

```
PCRE_SEARCH(PATTERN VARCHAR(1000), TEXT VARCHAR(4000), START INTEGER)
PCRE_SEARCH(PATTERN VARCHAR(1000), TEXT VARCHAR(4000))

RETURNS INTEGER
```

Description

PCRE searching function. Given a regular expression in **PATTERN**, and some text to search in **TEXT**, returns the 1-based position of the first match. **START** is an optional 1-based position from which to start the search (defaults to `1` if not specified). If no match is found, the function returns zero. If **PATTERN**, **TEXT**, or **START** is NULL, the result is NULL.

Parameters

PATTERN The Perl-compatible Regular Expression (PCRE) to search for

TEXT The text to search within

START The 1-based position from which to start the search. Defaults to `1` if omitted.

Examples

Simple searches showing the return value is a 1-based position or 0 in the case of failure:

```
VALUES
  (PCRE_SEARCH('FOO', 'FOOBAR')),
  (PCRE_SEARCH('BAR', 'FOOBAR')),
  (PCRE_SEARCH('BAZ', 'FOOBAR'))
```

```
1
----------
         1
         4
         0
```

A search to check whether a value looks vaguely like an IP address; note that the octets are not checked for 0-255 range:

```
VALUES PCRE_SEARCH('^\d{1,3}(\.\d{1,3}){3}$', '192.168.0.1')
```

```
1
----------
         1
```

A search demonstrating use of back-references to check that a closing tag matches the opening tag:

```
VALUES PCRE_SEARCH('<([A-Z][A-Z0-9]*)[^>]*>.*?</\1>', '<B>BOLD!</B>')
```

```
1
----------
         1
```

Searches demonstrating negative look-aheads:

```
VALUES
  (PCRE_SEARCH('Q(?!U)', 'QUACK')),
  (PCRE_SEARCH('Q(?!U)', 'QI'))
```

```
1
----------
         0
         1
```

See Also

- SQL source code
- C source code
- *PCRE_SUB scalar function*
- *PCRE_SPLIT table function*
- *PCRE_GROUPS table function*
- PCRE library homepage
- Wikipedia PCRE article

PCRE_SPLIT table function

Searches for all occurrences of regular expression **PATTERN** in **TEXT**, returning a table of all matches and the text between each match.

Prototypes

```
PCRE_SPLIT(PATTERN VARCHAR(1000), TEXT VARCHAR(4000))

RETURNS TABLE(
  ELEMENT INTEGER,
  SEPARATOR INTEGER,
  POSITION INTEGER,
  CONTENT VARCHAR(4000)
)
```

Description

PCRE string splitting function. Given a regular expression in **PATTERN**, and some text in **TEXT**, the function searches for every occurence of **PATTERN** in **TEXT** and breaks **TEXT** into chunks based on those matches. Each chunk is returned as a row in the result table which details whether or not the chunk was a result of a match, or text between the match.

Parameters

PATTERN The Perl-compatible Regular Expression (PCRE) to search for.

TEXT The text to search within.

Returns

ELEMENT The 1-based index of the chunk. Note that there are usually two rows for each index, one where *SEPARATOR* is zero and another where *SEPARATOR* is one. Therefore, one could consider the key of the result table to be (*ELEMENT*, *SEPARATOR*)

SEPARATOR Contains 1 if the row represents a match for **PATTERN**, and 0 if the row represents text between matches.

POSITION The 1-based position of *CONTENT* within the original **TEXT** parameter.

CONTENT The extract from **TEXT**.

Examples

An example demonstrating a simple split. Note that a row is still returned for the "missing" value, albeit with an empty *CONTENT* value:

```
SELECT
    T.ELEMENT,
    T.SEPARATOR,
    T.POSITION,
    T.CONTENT
FROM
    TABLE(
       PCRE_SPLIT(':', 'A:B:C::E')
    ) AS T
```

```
ELEMENT  SEPARATOR  POSITION  CONTENT
-------  ---------  --------  -------------------
      1          0         1  A
      1          1         2  :
      2          0         3  B
      2          1         4  :
      3          0         5  C
      3          1         6  :
      4          0         7
      4          1         7  :
      5          0         8  E
```

An example demonstrating a very rudimentary CSV parser. Note that to keep things simple, we actually treat the separator pattern as the data here, filter out the interleaved commas and remove the quotes surrounding delimited values:

```
SELECT
    T.ELEMENT,
    CASE WHEN LEFT(T.CONTENT, 1) = '"'
        THEN SUBSTR(T.CONTENT, 2, LENGTH(T.CONTENT) - 2)
        ELSE T.CONTENT
    END AS CONTENT
FROM
    TABLE(
        PCRE_SPLIT('([^",][^,]*|"[^"]*")', '"Some",CSV,",data"')
    ) AS T
WHERE
    T.SEPARATOR = 1
```

```
ELEMENT  CONTENT
-------  -------------------
      1  Some
      2  CSV
      3  ,data
```

See Also

- SQL source code
- C source code
- *PCRE_SEARCH scalar function*
- *PCRE_SUB scalar function*
- *PCRE_GROUPS table function*
- PCRE library homepage
- Wikipedia PCRE article

PCRE_SUB scalar function

Returns replacement pattern **REPL** with substitutions from matched groups of regular expression **PATTERN** in **TEXT** starting from 1-based **START**.

Prototypes

```
PCRE_SUB(PATTERN VARCHAR(1000), REPL VARCHAR(4000), TEXT VARCHAR(4000), START INTEGER)
PCRE_SUB(PATTERN VARCHAR(1000), REPL VARCHAR(4000), TEXT VARCHAR(4000))

RETURNS VARCHAR(4000)
```

Description

PCRE substitution function. Given a regular expression in **PATTERN**, a substitution pattern in **REPL**, some text to match in **TEXT**, and an optional 1-based **START** position for the search, returns **REPL** with backslash prefixed group specifications replaced by the corresponding matched group, e.g. `\0` refers to the group that matches the entire **PATTERN**, `\1` refers to the first capturing group in **PATTERN**. To include a literal backslash in **REPL** double it, i.e. `\\`. Returns NULL if the **PATTERN** does not match **TEXT**.

Note that ordinary C-style backslash escapes are *not* interpreted by this function within **REPL**, i.e. `\n` will *not* be replaced by a newline character. Use ordinary SQL hex-strings for this.

Parameters

PATTERN The Perl-Compatible Regular Expression (PCRE) to search for.

REPL The replacement pattern to return, after substitution of matched groups (indicated by back-slash prefixed numbers within this string).

TEXT The text to search within.

START The 1-based position from which to start the search. Defaults to 1 if omitted.

Examples

Simple searches demonstrating extraction of the matched portion of **TEXT** (if any):

```
VALUES
  (PCRE_SUB('FOO', '\0', 'FOOBAR')),
  (PCRE_SUB('FOO(BAR)?', '\0', 'FOOBAR')),
  (PCRE_SUB('BAZ', '\0', 'FOOBAR'))
```

```
1
-------------------...
FOO
FOOBAR
-
```

A substitution demonstrating the extraction of an IP address from some text:

```
VALUES PCRE_SUB('\b(\d{1,3}(\.\d{1,3}){3})\b', '\1', 'IP address: 192.168.0.1')
```

```
1
-----------------...
192.168.0.1
```

A substitution demonstrating the replacement of one HTML tag with another:

```
VALUES PCRE_SUB('<([A-Z][A-Z0-9]*)[^>]*>(.*?)</\1>', '<I>\2</I>', '<B>BOLD!</B>')
```

```
1
------------------...
<I>BOLD!</I>
```

A substitution demonstrating that look-aheads do not form part of the match:

```
VALUES PCRE_SUB('Q(?!U)', '\0', 'QI')
```

```
1
---------------...
Q
```

See Also

- SQL source code
- C source code
- *PCRE_SEARCH scalar function*
- *PCRE_SPLIT table function*
- *PCRE_GROUPS table function*
- PCRE library homepage
- Wikipedia PCRE article

PRIOR_DAYOFWEEK scalar function

Returns the latest date earlier than **ADATE**, which is also a particular day of the week, **ADOW** (1=Sunday, 2=Monday, 6=Saturday, etc.)

Prototypes

```
PRIOR_DAYOFWEEK(ADATE DATE, ADOW INTEGER)
PRIOR_DAYOFWEEK(ADATE TIMESTAMP, ADOW INTEGER)
PRIOR_DAYOFWEEK(ADATE VARCHAR(26), ADOW INTEGER)
PRIOR_DAYOFWEEK(ADOW INTEGER)

RETURNS DATE
```

Description

Returns the specified day of the week prior to the given date. Days of the week are specified in the same fashion as the built-in *DAYOFWEEK* function (i.e. 1=Sunday, 2=Monday, . . . 7=Saturday). If **ADATE** is omitted the current date is used.

Parameters

ADATE The date before which to return a specific day of the week. If this parameter is omitted the *CURRENT DATE* special register is used.

ADOW The day of the week to find specified as an integer where 1 represents Sunday, 2 is Monday, and so on.

Examples

Find the Monday before the start of 2010:

```
VALUES VARCHAR(PRIOR_DAYOFWEEK('2010-01-01', 2), ISO);
```

```
1
----------
2009-12-28
```

Find the last Friday in January, 2010:

```
VALUES VARCHAR(PRIOR_DAYOFWEEK(MONTHEND(2010, 1), 6), ISO);
```

```
1
----------
2010-01-29
```

See Also

- Source code
- *NEXT_DAYOFWEEK scalar function*

QUARTEREND scalar function

Returns the last day of the quarter that **ADATE** exists within, or the last day of the quarter **AQUARTER** in the year **AYEAR**.

Prototypes

```
QUARTEREND(AYEAR INTEGER, AQUARTER INTEGER)
QUARTEREND(ADATE DATE)
QUARTEREND(ADATE TIMESTAMP)
QUARTEREND(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the last day of **AQUARTER** in **AYEAR**, or the last day of the quarter of **ADATE** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AQUARTER** for which to return the ending date.

AQUARTER If provided, the quarter for which to return to the ending date.

ADATE If provided the date in the quarter for which to return the ending date. Either **AYEAR** and **AQUARTER**, or **ADATE** must be specified.

Examples

Calculate the ending date of the second quarter in 2010:

```
VALUES QUARTEREND(2010, 2);
```

```
1
----------
2010-06-30
```

Calculate the end date of the quarter containing the first of February, 2010:

```
VALUES QUARTEREND('2010-02-01');
```

```
1
----------
2010-03-31
```

See Also

- Source code

- *QUARTERSTART scalar function*
- QUARTER (built-in function)

QUARTERSTART scalar function

Returns the first day of the quarter that **ADATE** exists within, or the first day of the quarter **AQUARTER** in the year **AYEAR**.

Prototypes

```
QUARTERSTART(AYEAR INTEGER, AQUARTER INTEGER)
QUARTERSTART(ADATE DATE)
QUARTERSTART(ADATE TIMESTAMP)
QUARTERSTART(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the first day of **AQUARTER** in **AYEAR**, or the first day of the quarter of **ADATE** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AQUARTER** for which to return the starting date.

AQUARTER If provided, the quarter for which to return to the starting date.

ADATE If provided the date in the quarter for which to return the starting date. Either **AYEAR** and **AQUARTER**, or **ADATE** must be specified.

Examples

Calculate the starting date of the second quarter in 2010:

```
VALUES QUARTERSTART(2010, 2);
```

```
1
----------
2010-04-01
```

Calculate the start date of the quarter containing the first of February, 2010:

```
VALUES QUARTERSTART('2010-02-01');
```

```
1
----------
2010-01-01
```

See Also

- Source code
- *QUARTEREND scalar function*
- QUARTER (built-in function)

QUARTERWEEK scalar function

Returns the week of the quarter that **ADATE** exists within (weeks start on a Sunday, result will be in the range 1-14).

Prototypes

```
QUARTERWEEK(ADATE DATE)
QUARTERWEEK(ADATE TIMESTAMP)
QUARTERWEEK(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the week of the quarter of **ADATE**, where weeks start on a Sunday. The result will be in the range 1-14 as partial weeks are permitted. For example, if the first day of a quarter is a Saturday, it will be counted as week 1, which lasts one day. The next day, Sunday, will start week 2.

Parameters

ADATE The date to calculate the week of the quarter for.

Examples

Calculate the week of the quarter for 31st of January, 2010:

```
VALUES QUARTERWEEK(DATE(2010, 1, 31));
```

```
1
------
     6
```

Show the number of weeks in all quarters in the years 2007-2010:

```
SELECT YEAR(D) AS YEAR, QUARTER(D) AS QUARTER, QUARTERWEEK(QUARTEREND(D)) AS WEEKS
FROM TABLE(DATE_RANGE('2007-01-01', '2010-12-31', '300'));
```

```
YEAR        QUARTER     WEEKS
----------- ----------- ------
       2007           1     13
       2007           2     13
       2007           3     14
       2007          4     14
       2008          1     14
       2008          2     14
       2008          3     14
       2008          4     14
       2009          1     14
       2009          2     14
       2009          3     14
       2009          4     14
       2010          1     14
       2010          2     14
       2010          3     14
       2010          4     14
```

See Also

- Source code
- *QUARTERWEEK_ISO scalar function*

QUARTERWEEK_ISO scalar function

Returns the week of the quarter that **ADATE** exists within (weeks start on a Monday, result will be in the range 1-6).

Prototypes

```
QUARTERWEEK_ISO(ADATE DATE)
QUARTERWEEK_ISO(ADATE TIMESTAMP)
QUARTERWEEK_ISO(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the week of the quarter of **ADATE**, where weeks start on a Monday. The result will be in the range 1-14 as partial weeks are permitted. For example, if the first day of a month is a Sunday, it will be counted as week 1, which lasts one day. The next day, Monday, will start week 2.

Parameters

ADATE The date to calculate the week of the quarter for.

Examples

Calculate the week of the quarter for 31st of January, 2010:

```
VALUES QUARTERWEEK_ISO(DATE(2010, 1, 31));
```

```
1
------
     5
```

Show the number of weeks in all quarters in the years 2007-2010:

```
SELECT YEAR(D) AS YEAR, QUARTER(D) AS QUARTER, QUARTERWEEK_ISO(QUARTEREND(D)) AS WEEKS
FROM TABLE(DATE_RANGE('2007-01-01', '2010-12-31', '300'));
```

```
YEAR        QUARTER     WEEKS
----------- ----------- ------
       2007           1     13
       2007           2     14
       2007           3     14
       2007           4     14
       2008           1     14
       2008           2     14
       2008           3     14
       2008           4     14
       2009           1     14
       2009           2     14
       2009           3     14
       2009           4     14
       2010           1     14
       2010           2     14
       2010           3     14
       2010           4     14
```

See Also

- Source code
- *QUARTERWEEK scalar function*

QUOTE_IDENTIFIER scalar function

If **AIDENT** is an identifier which requires quoting, returns **AIDENT** surrounded by double quotes with all contained double quotes doubled. Useful when constructing SQL for EXECUTE IMMEDIATE within a procedure.

Prototypes

```
QUOTE_IDENTIFIER(AIDENT(VARCHAR(128))

RETURNS VARCHAR(258)
```

Description

Returns **AIDENT** surrounded by double quotes if **AIDENT** contains any characters which cannot appear in an identifier, as defined by the DB2 SQL dialect. Specifically this function is intended for correctly quoting SQL identifiers in generated SQL. Hence if **AIDENT** contains any lower-case, whitespace or symbolic characters, or begins with a numeral or underscore, it is returned quoted. If **AIDENT** contains no such characters it is returned verbatim.

Parameters

AIDENT The identifier to quote (if necessary).

Examples

Quote a simple identifier:

```
VALUES QUOTE_IDENTIFIER('MY_TABLE')
```

```
1
----------...
MY_TABLE
```

Quote an identifier containing characters that require quoting:

```
VALUES QUOTE_IDENTIFIER('MyTable')
```

```
1
-----------...
"MyTable"
```

Quote an identifier containing quotation marks:

```
VALUES QUOTE_IDENTIFIER('My "Table"')
```

```
1
-----------------...
"My ""Table"""
```

See Also

- Source code
- *QUOTE_STRING scalar function*

QUOTE_STRING scalar function

Returns **ASTRING** surrounded by single quotes with all necessary escaping. Useful when constructing SQL for EXECUTE IMMEDIATE within a procedure.

Prototypes

```
QUOTE_STRING(ASTRING VARCHAR(4000))

RETURNS VARCHAR(4000)
```

Description

Returns **ASTRING** surrounded by single quotes and performs any necessary escaping within the string to make it valid SQL. For example, single quotes within **ASTRING** are doubled, and control characters like CR or LF are returned as concatenated hex-strings.

Parameters

ASTRING The string to enclose in single-quotation marks.

Examples

Quote a simple string:

```
VALUES QUOTE_STRING('A string')
```

```
1
----------------...
'A string'
```

Quote a string containing an apostrophe (the delimiter for SQL strings):

```
VALUES QUOTE_STRING('Frank''s string')
```

```
1
---------------------...
'Frank''s string'
```

Quote a string containing a control character (in this case a line-feed):

```
VALUES QUOTE_STRING('A multi' || X'0A' || 'line string')
```

```
1
-------------------------------------...
'A multi' || X'0A' || 'line string'
```

See Also

- Source code
- *QUOTE_IDENTIFIER scalar function*

SECONDEND scalar function

Returns a TIMESTAMP at the end of **AHOUR:AMINUTE:ASECOND** on the date **AYEAR**, **AMONTH**, **ADAY**, or at the end of the second of **ATIMESTAMP**.

Prototypes

```
SECONDEND(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER, AHOUR INTEGER, AMINUTE INTEGER,
↪ ASECOND INTEGER)
SECONDEND(ATIMESTAMP TIMESTAMP)
SECONDEND(ATIMESTAMP VARCHAR(26))

RETURNS TIMESTAMP
```

Description

Returns a TIMESTAMP value representing the last microsecond of **ASECOND** in **AMINUTE** in **AHOUR** on the date given by **AYEAR**, **AMONTH**, and **ADAY**, or of the timestamp given by **ATIMESTAMP** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year component of the resulting timestamp.

AMONTH If provided, the month component of the resulting timestamp.

ADAY If provided, the day component of the resulting timestamp.

AHOUR If provided, the hour component of the resulting timestamp.

AMINUTE If provided, the minute component of the resulting timestamp.

ASECOND If provided, the second component of the resulting timestamp.

ATIMESTAMP If provided, the timestamp from which to derive the end of the second. Either **AYEAR**, **AMONTH**, **ADAY**, **AHOUR**, **AMINUTE**, and **ASECOND**, or **ATIMESTAMP** must be provided.

Examples

Round the specified timestamp up to one microsecond before the next second:

```
VALUES SECONDEND('2010-01-23 04:56:12.123456');
```

```
1
--------------------------
2010-01-23-04.56.12.999999
```

Generate a timestamp at the end of a second with the specified fields:

```
VALUES SECONDEND(2010, 2, 14, 9, 30, 44);
```

```
1
--------------------------
2010-02-14-09.30.44.999999
```

See Also

- Source code
- *SECONDSTART scalar function*
- SECOND (built-in function)

SECONDS scalar function

Returns an integer representation of the specified TIMESTAMP. The inverse of this function is *TIMESTAMP scalar function*.

Prototypes

```
SECONDS(ATIMESTAMP TIMESTAMP)
SECONDS(ATIMESTAMP DATE)
SECONDS(ATIMESTAMP VARCHAR(26))

RETURNS BIGINT
```

Description

Returns an integer representation of a TIMESTAMP. This function is a combination of the built-in *DAYS* and *MIDNIGHT_SECONDS* functions. The result is a BIGINT (64-bit integer value) representing the number of seconds since one day before 0001-01-01 at 00:00:00. The one day offset is due to the operation of the *DAYS* function.

Parameters

ATIMESTAMP The timestamp to convert to an integer representation. If a DATE is provided, then it will be treated as a TIMESTAMP with the equivalent date portion and a time portion of midnight.

Examples

Return an integer representation of the first instant of the year 2010:

```
VALUES SECONDS(YEARSTART(2010));
```

```
1
--------------------
         63397987200
```

Return the number of seconds in the year 2010:

```
VALUES SECONDS(YEARSTART(2011)) - SECONDS(YEARSTART(2010));
```

```
1
--------------------
            31536000
```

See Also

- Source code
- *TIMESTAMP scalar function*
- DAYS (built-in function)
- MIDNIGHT_SECONDS (built-in function)

SECONDSTART scalar function

Returns a TIMESTAMP at the start of **AHOUR:AMINUTE:ASECOND** on the date **AYEAR**, **AMONTH**, **ADAY**, or at the start of the second of **ATIMESTAMP**.

Prototypes

```
SECONDSTART(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER, AHOUR INTEGER, AMINUTE␣
↪INTEGER, ASECOND INTEGER)
SECONDSTART(ATIMESTAMP TIMESTAMP)
SECONDSTART(ATIMESTAMP VARCHAR(26))

RETURNS TIMESTAMP
```

Description

Returns a TIMESTAMP value representing the first microsecond of **ASECOND** in **AMINUTE** in **AHOUR** on the date given by **AYEAR**, **AMONTH**, and **ADAY**, or of the timestamp given by **ATIMESTAMP** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year component of the resulting timestamp.

AMONTH If provided, the month component of the resulting timestamp.

ADAY If provided, the day component of the resulting timestamp.

AHOUR If provided, the hour component of the resulting timestamp.

AMINUTE If provided, the minute component of the resulting timestamp.

ASECOND If provided, the second component of the resulting timestamp.

ATIMESTAMP If provided, the timestamp from which to derive the start of the second. Either **AYEAR**, **AMONTH**, **ADAY**, **AHOUR**, **AMINUTE**, and **ASECOND**, or **ATIMESTAMP** must be provided.

Examples

Truncate the specified timestamp to the nearest second:

```
VALUES SECONDSTART('2010-01-23 04:56:12.123456');
```

```
1
--------------------------
2010-01-23-04.56.12.000000
```

Generate a timestamp at the start of a second with the specified fields:

```
VALUES SECONDSTART(2010, 2, 14, 9, 30, 44);
```

```
1
--------------------------
2010-02-14-09.30.44.000000
```

See Also

- Source code
- *SECONDEND scalar function*
- SECOND (built-in function)

TABLE_COLUMNS scalar function

Returns a string containing the comma-separated list of columns of the specified table in the order they are defined

Prototypes

```
TABLE_COLUMNS(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1),
↪ INCLUDE_IDENTITY VARCHAR(1))
TABLE_COLUMNS(ATABLE VARCHAR(128), INCLUDE_GENERATED VARCHAR(1), INCLUDE_IDENTITY␣
↪VARCHAR(1))
TABLE_COLUMNS(ATABLE VARCHAR(128))
```

Description

This function returns a string containing a comma-separated list of the columns in the specified table in the order that they exist in the table.

If **ASCHEMA** is omitted it defaults to the value of the *CURRENT SCHEMA* special register. If the optional **INCLUDE_GENERATED** parameter is `'Y'` (the default), GENERATED ALWAYS columns will be included, otherwise they are excluded. GENERATED BY DEFAULT columns are always included. If the optional **INCLUDE_IDENTITY** parameter is `'Y'` (the default), IDENTITY columns will be included, otherwise they are excluded.

Parameters

ASCHEMA If provided, the schema containing the table for which to return a column list. Defaults to the value of the *CURRENT SCHEMA* special register if omitted.

ATABLE The table for which to return a column list.

INCLUDE_GENERATED If provided, specifies whether to include GENERATED ALWAYS columns in the result. Defaults to `'Y'` if omitted.

INCLUDE_IDENTITY If provided, specifies whether to include IDENTITY columns in the result. Defaults to `'Y'` if omitted.

Examples

Return a comma-separated list of the columns in the *SYSIBM.SYSTABLES* table:

```
VALUES TABLE_COLUMNS('SYSIBM', 'SYSTABLES', 'Y', 'Y');
```

```
1
---------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------
NAME,CREATOR,TYPE,CTIME,REMARKS,PACKED_DESC,VIEW_DESC,COLCOUNT,FID,TID,CARD,NPAGES,
↪FPAGES,OVERFLOW,PARENTS,CHILDREN,SELFREFS,KEYCOLUMNS,KEYOBID,REL_DESC,BASE_NAME,
↪BASE_SCHEMA,TBSPACE,INDEX_TBSPACE,LONG_TBSPACE,KEYUNIQUE,CHECKCOUNT,CHECK_DESC,
↪STATS_TIME,DEFINER,TRIG_DESC,DATA_CAPTURE,STATUS,CONST_CHECKED,PMAP_ID,ENCODING_
↪SCHEME,PCTFREE,ROWTYPESCHEMA,ROWTYPENAME,APPEND_MODE,PARTITION_MODE,REFRESH,REFRESH_
↪TIME,LOCKSIZE,VOLATILE,REMOTE_DESC,CLUSTERED,AST_DESC,DROPRULE,LOGINDEXBUILD,
↪PROPERTY,STATISTICS_PROFILE,COMPRESSION,ACCESS_MODE,ACTIVE_BLOCKS,
↪MAXFREESPACESEARCH,AVGCOMPRESSEDROWSIZE,AVGROWCOMPRESSIONRATIO,AVGROWSIZE,
↪PCTROWSCOMPRESSED,CODEPAGE,PCTPAGESSAVED,LAST_REGEN_TIME,SECPOLICYID,
↪PROTECTIONGRANULARITY,INVALIDATE_TIME,DEFINERTYPE,ALTER_TIME,AUDITPOLICYID,
↪COLLATIONID,COLLATIONID_ORDERBY,ONCOMMIT,ONROLLBACK,LOGGED,LASTUSED
```

See the implementation of *EXPORT_TABLE scalar function* for an example of the usage of this function within a stored procedure.

See Also

- Source code
- *EXPORT_TABLE scalar function*
- *LOAD_TABLE scalar function*
- SYSCAT.COLUMNS (built-in catalogue view)

TIME scalar function

Constructs a TIME from the specified hours, minutes and seconds, or seconds from midnight.

Prototypes

```
TIME(AHOUR INTEGER, AMINUTE INTEGER, ASECOND INTEGER)
TIME(ASECONDS BIGINT)
TIME(ASECONDS INTEGER)

RETURNS TIME
```

Description

Returns a TIME with the components specified by **AHOUR**, **AMINUTE** and **ASECOND** in the first case. In the second case, returns a TIME **ASECONDS** after midnight. If **ASECONDS** represents a period longer than a day, the value used is **ASECONDS** mod `86400` (the "date" portion of the seconds value is removed before calculation). This function is essentially the reverse of the *MIDNIGHT_SECONDS* function.

Parameters

AHOUR If provided, specifies the hour component of the resulting TIME.

AMINUTE If provided, specifies the minute component of the resulting TIME.

ASECONDS If **AHOUR** and **AMINUTE** are provided, specifies the second component of the resulting TIME. Otherwise, specifies the number of seconds after minute from which the hour and minute components will be derived.

Examples

Construct a time representing midnight:

```
VALUES TIME(0);
```

```
1
--------
00:00:00
```

Construct a time representing half past noon:

```
VALUES TIME(12, 30, 0);
```

```
1
--------
12:30:00
```

See Also

- Source code
- *DATE scalar function*
- TIME (built-in function)
- MIDNIGHT_SECONDS (built-in function)

TIMESTAMP scalar function

Constructs a TIMESTAMP from the specified seconds after the epoch. This is the inverse of *SECONDS scalar function*.

Prototypes

```
TIMESTAMP(ASECONDS BIGINT)
TIMESTAMP(AYEAR INTEGER, AMONTH INTEGER, ADAY INTEGER, AHOUR INTEGER, AMINUTE INTEGER,
↪ ASECOND INTEGER, AMICROSECOND INTEGER)

RETURNS TIMESTAMP
```

Description

The first version of this function returns a TIMESTAMP **ASECONDS** seconds after 0000-12-31 00:00:00. This function is essentially the reverse of *SECONDS scalar function*. The **ASECONDS** value MUST be greater than `86400` (it must include a "date" portion) otherwise the returned value has an invalid year of 0000 and an error will occur.

The second version of this function simply constructs a timestamp from the given integer fields.

Parameters

ASECONDS The number of seconds after the epoch (0000-12-31 00:00:00) which the resulting TIMESTAMP will represent.

AYEAR The year for the resulting timestamp.

AMONTH The month for the resulting timestamp (1-12).

ADAY The day for the resulting timestamp (1-31).

AHOUR The hours for the resulting timestamp (0-23).

AMINUTE The minutes for the resulting timestamp (0-59).

ASECOND The seconds for the resulting timestamp (0-59).

AMICROSECOND The microseconds for the resulting timestamp (0-999999).

Examples

Construct a TIMESTAMP representing the epoch (note that 0 cannot be used due to the offset mentioned in *SECONDS scalar function*:

```
VALUES TIMESTAMP(86400);
```

```
1
--------------------------
0001-01-01-00.00.00.000000
```

Calculate a TIMESTAMP 10 seconds before midnight on new year's day 2000 (admittedly this would be more simply accomplished with `TIMESTAMP(YEARSTART(2000)) - 10 SECONDS`, but for the sake of demonstration we're using a round-trip of TIMESTAMP and *SECONDS scalar function* here):

```
VALUES TIMESTAMP(SECONDS(YEARSTART(2000)) - 10);
```

```
1
--------------------------
1999-12-31-23.59.50.000000
```

Construct a timestamp from a set of literal values:

```
VALUES TIMESTAMP(2000, 1, 1, 0, 0, 0, 0);
```

```
1
--------------------------
2000-01-01-00.00.00.000000
```

See Also

- Source code
- *SECONDS scalar function*
- TIMESTAMP (built-in function)

TS_FORMAT scalar function

A version of C's strftime() for DB2. Formats **ATIMESTAMP** according to the **AFORMAT** string, containing %-prefixed templates which will be replaced with elements of **ATIMESTAMP**.

Prototypes

```
TS_FORMAT(AFORMAT VARCHAR(100), ATIMESTAMP TIMESTAMP)
TS_FORMAT(AFORMAT VARCHAR(100), ATIMESTAMP DATE)
TS_FORMAT(AFORMAT VARCHAR(100), ATIMESTAMP TIME)
TS_FORMAT(AFORMAT VARCHAR(100), ATIMESTAMP VARCHAR(26))

RETURNS VARCHAR(100)
```

Description

TS_FORMAT is a reimplementation of C's strftime() function which converts a TIMESTAMP (or DATE, TIME, or VARCHAR(26) containing a string representation of a TIMESTAMP) into a VARCHAR according to a format string containing %-prefixed templates which will be replaced with components derived from the provided TIMESTAMP. The templates which can be used within the format string are as follows:

Template	Meaning
%a	Locale's abbreviated weekday name
%A	Locale's full weekday name

Continued on next page

Table 1.1 – continued from previous page

Template	Meaning
%b	Locale's abbreviated month name
%B	Locale's full month name
%c	Locale's appropriate date and time representation
%C	The century number (year/100), 00-99
%d	Day of the month as a decimal number, 01-31
%D	Equivalent to `'%m/%d/%y'` (US format)
%e	Like %d, but with leading space instead of zero
%F	Equivalent to `'%Y-%m-%d'` (ISO8601 format)
%G	ISO8601 year with century as a decimal number
%g	ISO8601 year without century as a decimal number
%h	Half of the year as a decimal number, 1-2
%H	Hour (24-hr clock) as a decimal number, 00-23
%I	Hour (12-hr clock) as a decimal number, 01-12
%j	Day of the year as a decimal number, 001-366
%k	Like %H with leading space instead of zero
%l	Like %I with leading space instead of zero
%m	Month as a decimal number, 01-12
%M	Minute as a decimal number, 00-59
%n	Newline character (`X'0A'`)
%p	Locale's equivalent of either AM or PM
%P	Like `'%p'` but lowercase
%q	Quarter of the year as decimal number, 1-4
%S	Second as a decimal number, 00-61
%t	A tab character (`X'09'`)
%T	Equivalent to `'%H:%M:%S'`
%u	Weekday as a decimal number, 1 (Monday) - 7 (Sunday)
%U	Week number of the year (Sunday as the first day of the week) as a decimal number, 01-54
%V	ISO8601 Week number of the year (Monday as the first day of the week) as a decimal number, 01-53
%w	Weekday as a decimal number, 1 (Sunday) - 7 (Monday)
%W	Equivalent to `'%V'`
%x	Locale's appropriate date representation
%X	Locale's appropriate time representation
%y	Year without century as a decimal number, 00-99
%Y	Year with century as a decimal number
%Z	Time zone offset (no characters if no time zone exists)
%%	A literal % character

Note: This routine was primarily included in response to the rather useless TIMESTAMP_FORMAT included in early versions (pre-fixpack 4?) of DB2 9.5, which only permitted specification of a single ISO8601-ish format string. Later fixpacks and DB2 9.7 now include a fairly decent TIMESTAMP_FORMAT implementation which is considerably more efficient than this one, although still somewhat limited in the range of available templates.

Parameters

AFORMAT A string containing the templates to substitute with the fields of **ATIMESTAMP**.

ATIMESTAMP A TIMESTAMP, DATE, TIME, or VARCHAR(26) value (containing a string representation of a timestamp) which will be used to calculate the substitutions for the templates in **AFORMAT**.

Examples

Format the 7th of August, 2010 in US style:

```
VALUES TS_FORMAT('%m/%d/%Y', '2010-08-07');
```

```
1
--------------------------------------------------------------------------------------
↪--------------
08/07/2010
```

Construct a sentence describing the week of a given date:

```
VALUES TS_FORMAT('Week %U of %B, %Y', '2010-01-01');
```

```
1
--------------------------------------------------------------------------------------
↪--------------
Week 01 of January, 2010
```

See Also

- Source code
- TIMESTAMP_FORMAT (built-in function)

UNICODE_REPLACE_BAD function

Returns **SOURCE** with characters that are invalid in UTF-8 encoding replaced with the string **REPL**.

Prototypes

```
UNICODE_REPLACE_BAD(SOURCE VARCHAR(4000), REPL VARCHAR(100))
UNICODE_REPLACE_BAD(SOURCE VARCHAR(4000))

RETURNS VARCHAR(4000)
```

Description

Under certain circumstances, DB2 will permit text containing characters invalid in the UTF-8 encoding scheme to be inserted into a column intended to contain UTF-8 encoded data. While this doesn't cause a problem for DB2 queries, it can cause issues for down-stream appliations. This function provides a means of stripping or replacing such invalid characters.

Parameters

SOURCE The string to search for characters invalid in the UTF-8 encoding scheme.

REPL The string to replace any invalid sequences with. Defaults to the empty string if omitted.

Examples

Replacement of truncated UTF-8 characters:

```
VALUES
     (UNICODE_REPLACE_BAD('FOO' || X'C2', 'BAR'))
```

```
1
--------------------....
FOOBAR
```

Replacement of invalid characters in the middle of a string:

```
VALUES
     (UNICODE_REPLACE_BAD('FOO' || X'80' || BAR))
```

```
1
--------------------....
FOOBAR
```

See Also

- SQL source code
- C source code
- Wikipedia UTF-8 article

WEEKEND scalar function

Returns the last day (always a Saturday) of the week that **ADATE** exists within, or the last day of the week **AWEEK** in the year **AYEAR**.

Prototypes

```
WEEKEND(AYEAR INTEGER, AWEEK INTEGER)
WEEKEND(ADATE DATE)
WEEKEND(ADATE TIMESTAMP)
WEEKEND(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the last day of **AWEEK** in **AYEAR**, or the last day of the week of **ADATE** (always a Saturday) depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AWEEK** for which to return the ending date.

AWEEK If provided, the week for which to return to the ending date.

ADATE If provided the date in the week for which to return the ending date. Either **AYEAR** and **AWEEK**, or **ADATE** must be specified.

Examples

Calculate the ending date of the last week in 2010:

```
VALUES WEEKEND(2010, WEEKSINYEAR(2010));
```

```
1
----------
2011-01-01
```

Calculate the end of the week for the 28th of January, 2009:

```
VALUES WEEKEND('2009-01-28');
```

```
1
----------
2009-01-31
```

See Also

- Source code
- *WEEKSTART scalar function*
- *WEEKEND_ISO scalar function*
- WEEK (built-in function)

WEEKEND_ISO scalar function

Returns the last day (always a Sunday) of the week that **ADATE** exists within, or the last day of the week **AWEEK** in the year **AYEAR** according to the ISO8601 standard.

Prototypes

```
WEEKEND_ISO(AYEAR INTEGER, AWEEK INTEGER)
WEEKEND_ISO(ADATE DATE)
WEEKEND_ISO(ADATE TIMESTAMP)
WEEKEND_ISO(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the last day of **AWEEK** in **AYEAR** according to the ISO8601 standard, or the last day of the week of **ADATE** (always a Sunday) depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AWEEK** for which to return the ending date.

AWEEK If provided, the week for which to return to the ending date.

ADATE If provided the date in the week for which to return the ending date. Either **AYEAR** and **AWEEK**, or **ADATE** must be specified.

Examples

Calculate the ending date of the last week in 2010:

```
VALUES WEEKEND_ISO(2010, WEEKSINYEAR_ISO(2010));
```

```
1
----------
2011-01-02
```

Calculate the end of the week for the 28th of January, 2009:

```
VALUES WEEKEND_ISO('2009-01-28');
```

```
1
----------
2009-02-01
```

See Also

- Source code
- *WEEKSTART_ISO scalar function*
- *WEEKEND scalar function*
- WEEK_ISO (built-in function)

WEEKSINMONTH scalar function

Returns the number of weeks within the month that **ADATE** exists within, or the number of weeks in **AMONTH** in **AYEAR**.

Prototypes

```
WEEKSINMONTH(AYEAR INTEGER, AMONTH INTEGER)
WEEKSINMONTH(ADATE DATE)
WEEKSINMONTH(ADATE TIMESTAMP)
WEEKSINMONTH(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the number of weeks in **AMONTH** in **AYEAR** (weeks start on a Sunday, and partial weeks are permitted at the start and end of the month), or the number of weeks in the month that **ADATE** exists within depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year containing **AMONTH** for which to calculate the number of weeks.

AMONTH If provided, the month within **AYEAR** for which to calculate the number of weeks.

ADATE If provided, the date within the month for which to calculate the number of weeks. Either **AYEAR** and **AMONTH**, or **ADATE** must be provided.

Examples

Calculate the number of weeks in January 2010:

```
VALUES WEEKSINMONTH(2010, 1);
```

```
1
------
     6
```

Calculate the number of weeks in the months of 2010:

```
SELECT MONTH(D) AS MONTH, WEEKSINMONTH(D) AS WEEKS
FROM TABLE(DATE_RANGE('2010-01-01', '2010-12-01', 100));
```

```
MONTH       WEEKS
----------- ------
          1      6
          2      5
          3      5
          4      5
          5      6
          6      5
          7      5
          8      5
          9      5
         10      6
         11      5
         12      5
```

See Also

- Source code
- *WEEKSINMONTH_ISO scalar function*
- MONTH (built-in function)
- WEEK (built-in function)

WEEKSINMONTH_ISO scalar function

Returns the number of weeks within the month that **ADATE** exists within, or the number of weeks in **AMONTH** in **AYEAR**.

Prototypes

```
WEEKSINMONTH_ISO(AYEAR INTEGER, AMONTH INTEGER)
WEEKSINMONTH_ISO(ADATE DATE)
WEEKSINMONTH_ISO(ADATE TIMESTAMP)
WEEKSINMONTH_ISO(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the number of weeks in **AMONTH** in **AYEAR** (weeks start on a Monday, and partial weeks are permitted at the start and end of the month), or the number of weeks in the month that **ADATE** exists within depending on the variant of the function that is called.

Note: As far as I'm aware, ISO8601 doesn't say anything about weeks within a month, hence why this function differs from *WEEKSINYEAR_ISO scalar function* which does *not* permit partial weeks at the start and end of a year. This function simply mirrors the functionality of *WEEKSINMONTH scalar function* but with a definition of weeks that start on a Monday instead of Sunday.

Parameters

AYEAR If provided, the year containing **AMONTH** for which to calculate the number of weeks.

AMONTH If provided, the month within **AYEAR** for which to calculate the number of weeks.

ADATE If provided, the date within the month for which to calculate the number of weeks. Either **AYEAR** and **AMONTH**, or **ADATE** must be provided.

Examples

Calculate the number of weeks in January 2010:

```
VALUES WEEKSINMONTH_ISO(2010, 1);
```

```
1
------
     5
```

Calculate the number of weeks in the months of 2010:

```
SELECT MONTH(D) AS MONTH, WEEKSINMONTH_ISO(D) AS WEEKS
FROM TABLE(DATE_RANGE('2010-01-01', '2010-12-01', 100));
```

```
MONTH       WEEKS
----------- ------
          1      5
          2      4
          3      5
          4      5
          5      6
          6      5
          7      5
          8      6
          9      5
         10      5
         11      5
         12      5
```

See Also

- Source code
- *WEEKSINMONTH scalar function*
- MONTH (built-in function)
- WEEK_ISO (built-in function)

WEEKSINYEAR scalar function

Returns the number of weeks within the year that **ADATE** exists within, or the number of weeks in **AYEAR**.

Prototypes

```
WEEKSINYEAR(AYEAR INTEGER)
WEEKSINYEAR(ADATE DATE)
WEEKSINYEAR(ADATE TIMESTAMP)
WEEKSINYEAR(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the number of weeks in **AYEAR** (weeks start on a Sunday, and partial weeks are permitted at the start and end of the year), or the number of weeks in the year that **ADATE** exists within depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year for which to calculate the number of weeks.

ADATE If provided, the date in the year for which to calculate the number of weeks. Either **AYEAR** or **ADATE** must be specified.

Examples

Calculate the number of weeks in the year 2010:

```
VALUES WEEKSINYEAR(2010);
```

```
1
------
    53
```

Calculate the number of weeks in the first 10 years of the 21st century:

```
SELECT YEAR(D) AS YEAR, WEEKSINYEAR(D) AS WEEKS
FROM TABLE(DATE_RANGE('2000-01-01', '2010-01-01', 10000));
```

```
YEAR        WEEKS
----------- ------
       2000     54
       2001     53
       2002     53
       2003     53
       2004     53
       2005     53
       2006     53
       2007     53
       2008     53
       2009     53
       2010     53
```

See Also

- Source code
- *WEEKSINYEAR_ISO scalar function*
- WEEK (built-in function)

WEEKSINYEAR_ISO scalar function

Returns the number of weeks within the year that **ADATE** exists within, or the number of weeks in **AYEAR** according to the ISO8601 standard.

Prototypes

```
WEEKSINYEAR_ISO(AYEAR INTEGER)
WEEKSINYEAR_ISO(ADATE DATE)
WEEKSINYEAR_ISO(ADATE TIMESTAMP)
WEEKSINYEAR_ISO(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the number of weeks in **AYEAR** according to the ISO8601 standard (weeks start on a Monday, and overlap calendar year ends to ensure all weeks are "whole"), or the number of weeks in the year that **ADATE** exists within depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year for which to calculate the number of weeks.

ADATE If provided, the date in the year for which to calculate the number of weeks. Either **AYEAR** or **ADATE** must be specified.

Examples

Calculate the number of weeks in the year 2010 according to ISO8601:

```
VALUES WEEKSINYEAR_ISO(2010);
```

```
1
------
    52
```

Calculate the number of weeks in the first 10 years of the 21st century according to ISO8601:

```
SELECT YEAR(D) AS YEAR, WEEKSINYEAR_ISO(D) AS WEEKS
FROM TABLE(DATE_RANGE('2000-01-01', '2010-01-01', 10000));
```

```
YEAR        WEEKS
----------- ------
       2000     52
       2001     52
       2002     52
       2003     52
       2004     53
       2005     52
       2006     52
       2007     52
       2008     52
       2009     53
       2010     52
```

See Also

- Source code
- *WEEKSINYEAR scalar function*
- WEEK_ISO (built-in function)

WEEKSTART scalar function

Returns the first day (always a Sunday) of the week that **ADATE** exists within, or the first day of the week **AWEEK** in the year **AYEAR**.

Prototypes

```
WEEKSTART(AYEAR INTEGER, AWEEK INTEGER)
WEEKSTART(ADATE DATE)
WEEKSTART(ADATE TIMESTAMP)
WEEKSTART(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the first day of **AWEEK** in **AYEAR**, or the first day of the week of **ADATE** (always a Sunday) depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AWEEK** for which to return the starting date.

AWEEK If provided, the week for which to return to the starting date.

ADATE If provided the date in the week for which to return the starting date. Either **AYEAR** and **AWEEK**, or **ADATE** must be specified.

Examples

Calculate the starting date of the first week in 2010:

```
VALUES WEEKSTART(2010, 1);
```

```
1
----------
2009-12-27
```

Calculate the start of the week for the 28th of January, 2009:

```
VALUES WEEKSTART('2009-01-28');
```

```
1
----------
2009-01-25
```

See Also

- Source code
- *WEEKEND scalar function*
- *WEEKSTART_ISO scalar function*
- WEEK (built-in function)

WEEKSTART_ISO scalar function

Returns the first day (always a Monday) of the week that **ADATE** exists within, or the first day of the week **AWEEK** in the year **AYEAR** according to the ISO8601 standard.

Prototypes

```
WEEKSTART_ISO(AYEAR INTEGER, AWEEK INTEGER)
WEEKSTART_ISO(ADATE DATE)
WEEKSTART_ISO(ADATE TIMESTAMP)
WEEKSTART_ISO(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the first day of **AWEEK** in **AYEAR** according to the ISO8601 standard, or the first day of the week of **ADATE** (always a Monday) depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year of **AWEEK** for which to return the starting date.

AWEEK If provided, the week for which to return to the starting date.

ADATE If provided the date in the week for which to return the starting date. Either **AYEAR** and **AWEEK**, or **ADATE** must be specified.

Examples

Calculate the starting date of the first week in 2010:

```
VALUES WEEKSTART_ISO(2010, 1);
```

```
1
----------
2010-01-04
```

Calculate the start of the week for the 28th of January, 2009:

```
VALUES WEEKSTART_ISO('2009-01-28');
```

```
1
----------
2009-01-26
```

See Also

- Source code
- *WEEKEND_ISO scalar function*
- *WEEKSTART scalar function*
- WEEK_ISO (built-in function)

WORKINGDAY scalar function

Calculates the working day of a specified date relative to another date which defaults to the start of the month

Prototypes

```
WORKINGDAY(ADATE DATE, RELATIVE_TO DATE, ALOCATION VARCHAR(10))
WORKINGDAY(ADATE DATE, RELATIVE_TO DATE)
WORKINGDAY(ADATE DATE, ALOCATION VARCHAR(10))
WORKINGDAY(ADATE DATE)

RETURNS INTEGER
```

Description

The WORKINGDAY function calculates the working day of a specified date relative to another date. The working day is defined as the number of days which are not Saturday or Sunday from the starting date to the specified date, plus one. Hence, if the starting date is neither a Saturday nor a Sunday, it is working day 1, the next non-weekend-day is working day 2 and so on.

Requesting the working day of a Saturday or a Sunday will return the working day value of the prior Friday; it is not an error to query the working day of a weekend day, you should instead check for this in the calling code.

If the **RELATIVE_TO** parameter is omitted it will default to the start of the month of the **ADATE** parameter. In other words, by default this function calculates the working day of the month of a given date.

If you wish to take into account more than merely weekend days when calculating working days, insert values into VACATIONS. If a vacation date occurs between the starting date and the target date (inclusive), it will count as another weekend date resulting in a working day one less than would otherwise be calculated. Note that the VACATIONS table will only be used when you specify a value for the optional **ALOCATION** parameter. This parameter is used to filter the content of the VACATIONS table under the assumption that different locations, most likely countries, will have different public holidays.

Parameters

ADATE The date to calculate the working day from.

RELATIVE_TO If specified, the date to calculate the working day relative to, i.e. the function counts the number of working days between **RELATIVE_TO** and **ADATE**. If omitted, defaults to the start of the month of **ADATE**.

ALOCATION If specified, causes the function to take into account additional vacation days defined in VACATIONS with the specified *LOCATION*.

Examples

Calculate the working day of the first date in 2010:

```
VALUES WORKINGDAY(YEARSTART(2010));
```

```
1
-----------
          1
```

Calculate the working day of the 4th of January, 2010 (the 2nd and 3rd of January 2010 are Saturday and Sunday respectively):

```
VALUES WORKINGDAY(DATE(2010, 1, 4))
```

```
1
-----------
          2
```

Calculate the number of working days in January 2010:

```
VALUES WORKINGDAY(MONTHEND(2010, 1))
```

```
1
-----------
         21
```

Calculate the total number of working days in 2010:

```
VALUES WORKINGDAY(YEAREND(2010), YEARSTART(2010))
```

```
1
-----------
        261
```

See Also

- Source code

YEAR_ISO scalar function

Returns the year of **ADATE**, unless the ISO week of **ADATE** exists in the prior year in which case that year is returned.

Prototypes

```
YEAR_ISO(ADATE DATE)
YEAR_ISO(ADATE TIMESTAMP)
YEAR_ISO(ADATE VARCHAR(26))

RETURNS SMALLINT
```

Description

Returns the year of **ADATE**, unless the ISO week number (see the built-in function WEEK_ISO) of **ADATE** belongs to the prior year, in which case the prior year is returned.

Parameters

ADATE The date to calculate the ISO-week based year number for.

Examples

Calculate the ISO-week based year number of the 1st of January, 2010:

```
VALUES YEAR_ISO(DATE(2010, 1, 1));
```

```
1
------
  2009
```

Calculate the ISO-week based year number of the 4th of January, 2010 (dates beyond the 4th of January will always be in the year of the date given the definition of ISO weeks):

```
VALUES YEAR_ISO(DATE(2010, 1, 4)));
```

```
1
------
  2010
```

See Also

- Source code
- YEAR (built-in function)
- WEEK_ISO (built-in function)

YEAREND scalar function

Returns the last day of the year **AYEAR**, or the last day of the year of **ADATE**.

Prototypes

```
YEAREND(AYEAR INTEGER)
YEAREND(ADATE DATE)
YEAREND(ADATE TIMESTAMP)
YEAREND(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the last day of **AYEAR**, or the last day of the year of **ADATE** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year for which to return the ending date.

ADATE If provided the date in the year for which to return the ending date. Either **AYEAR** or **ADATE** must be specified.

Examples

Calculate the ending date of 2010:

```
VALUES YEAREND(2010);
```

```
1
----------
2010-12-31
```

Calculate the ending date of the year for the 28th February, 2009:

```
VALUES YEAREND('2009-02-28');
```

```
1
----------
2009-12-31
```

See Also

- Source code
- *YEARSTART scalar function*
- YEAR (built-in function)

YEARSTART scalar function

Returns the first day of the year that **ADATE** exists within, or the first day of the year **AYEAR**.

Prototypes

```
YEARSTART(AYEAR INTEGER)
YEARSTART(ADATE DATE)
YEARSTART(ADATE TIMESTAMP)
YEARSTART(ADATE VARCHAR(26))

RETURNS DATE
```

Description

Returns a DATE representing the first day of **AYEAR**, or the first day of the year of **ADATE** depending on the variant of the function that is called.

Parameters

AYEAR If provided, the year for which to return the starting date.

ADATE If provided the date in the year for which to return the starting date. Either **AYEAR** or **ADATE** must be specified.

Examples

Calculate the starting date of 2010:

```
VALUES YEARSTART(2010);
```

```
1
----------
2010-01-01
```

Calculate the starting date of the year for the 28th February, 2009:

```
VALUES YEARSTART('2009-02-28');
```

```
1
----------
2009-01-01
```

See Also

- Source code
- *YEAREND scalar function*
- YEAR (built-in function)

1.7.2 Procedures

ASSERT_COLUMN_EXISTS procedure

Raises an assertion error if the specified column doesn't exist.

Prototypes

```
ASSERT_COLUMN_EXISTS(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128), ACOLNAME VARCHAR(128))
ASSERT_COLUMN_EXISTS(ATABLE VARCHAR(128), ACOLNAME VARCHAR(128))
```

Description

Raises the ASSERT_FAILED_STATE state if **ACOLNAME** does not exist in the table specified by **ASCHEMA** and **ATABLE**. If not specified, **ASCHEMA** defaults to the value of the *CURRENT SCHEMA* special register.

Parameters

ASCHEMA Specifies the schema containing the table to check. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATABLE Specifies the name of the table to check.

ACOLNAME Specifies the name of the column to test for existence.

Examples

Test the TABNAME column exists in the SYSCAT.TABLES view:

```
CALL ASSERT_COLUMN_EXISTS('SYSCAT', 'TABLES', 'TABNAME');
```

Test the existence of a made-up column in the SYSCAT.TABLES view:

```
CALL ASSERT_COLUMN_EXISTS('SYSCAT', 'TABLES', 'FOO');
```

```
SQL0438N  Application raised error or warning with diagnostic text: "FOO
does not exist in SYSCAT.TABLES                 ".  SQLSTATE=90001
```

See Also

- Source code
- *ASSERT_TABLE_EXISTS procedure*
- *ASSERT_TRIGGER_EXISTS procedure*
- *ASSERT_ROUTINE_EXISTS procedure*
- ASSERT_FAILED_STATE

ASSERT_SIGNALS procedure

Signals ASSERT_FAILED_STATE if the execution of **SQL** doesn't signal SQLSTATE **STATE**, or signals a different SQLSTATE.

Prototypes

```
ASSERT_SIGNALS(STATE CHAR(5), SQL CLOB(2M))
```

Description

Raises the ASSERT_FAILED_STATE if executing **SQL** does NOT raise SQLSTATE **STATE**. **SQL** must be capable of being executed by EXECUTE IMMEDIATE, i.e. no queries or SIGNAL calls.

Parameters

STATE The SQLSTATE that is expected to be raised by executing the content of the **SQL** parameter.

SQL The SQL statement to execute.

Examples

Attempt to drop the non-existent table FOO, and confirm that the operation raises SQLSTATE 42704:

```
CALL ASSERT_SIGNALS('42704', 'DROP TABLE FOO');
```

Raise the ASSERT_FAILED_STATE by attempting to assert that the same SQLSTATE is raised by simply querying the current date:

```
CALL ASSERT_SIGNALS('42704', 'VALUES CURRENT DATE');
```

```
SQL0438N  Application raised error or warning with diagnostic text: "VALUES
CURRENT DATE  signalled SQLSTATE 00000 instead of 42704".  SQLSTATE=90001
```

See Also

- Source code

ASSERT_ROUTINE_EXISTS procedure

Raises an assertion error if the specified function or stored procedure doesn't exist.

Prototypes

```
ASSERT_ROUTINE_EXISTS(ASCHEMA VARCHAR(128), AROUTINE VARCHAR(128))
ASSERT_ROUTINE_EXISTS(AROUTINE VARCHAR(128))
```

Description

Raises the ASSERT_FAILED_STATE state if the function or stored procedure specified by **ASCHEMA** and **AROUTINE** does not exist. If not specified, **ASCHEMA** defaults to the value of the *CURRENT SCHEMA* special register.

Parameters

ASCHEMA Specifies the schema containing the routine to check. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATRIGGER Specifies the name of the routine to check.

Examples

Test the *UTILS.DATE* function exists:

```
CALL ASSERT_ROUTINE_EXISTS('UTILS', 'DATE');
```

Test the existence of the routine *FOO* in the current schema:

```
CALL ASSERT_ROUTINE_EXISTS('FOO');
```

```
SQL0438N  Application raised error or warning with diagnostic text:
"DB2INST1.FOO                                      does not exist".
SQLSTATE=90001
```

See Also

- Source code
- *ASSERT_COLUMN_EXISTS procedure*
- *ASSERT_TABLE_EXISTS procedure*
- *ASSERT_ROUTINE_EXISTS procedure*
- ASSERT_FAILED_STATE

ASSERT_TABLE_EXISTS procedure

Raises an assertion error if the specified table doesn't exist.

Prototypes

```
ASSERT_TABLE_EXISTS(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128))
ASSERT_TABLE_EXISTS(ATABLE VARCHAR(128))
```

Description

Raises the ASSERT_FAILED_STATE state if the table or view specified by **ASCHEMA** and **ATABLE** does not exist. If not specified, **ASCHEMA** defaults to the value of the *CURRENT SCHEMA* special register.

Parameters

ASCHEMA Specifies the schema containing the table to check. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATABLE Specifies the name of the table to check.

Examples

Test the SYSCAT.TABLES view exists:

```
CALL ASSERT_TABLE_EXISTS('SYSCAT', 'TABLES');
```

Test the existence of a made-up table in SYSCAT:

```
CALL ASSERT_TABLE_EXISTS('SYSCAT', 'FOO');
```

```
SQL0438N  Application raised error or warning with diagnostic text:
"SYSCAT.FOO                                        does not exist".
SQLSTATE=90001
```

See Also

- Source code
- *ASSERT_COLUMN_EXISTS procedure*
- *ASSERT_TRIGGER_EXISTS procedure*
- *ASSERT_ROUTINE_EXISTS procedure*
- ASSERT_FAILED_STATE

ASSERT_TRIGGER_EXISTS procedure

Raises an assertion error if the specified trigger doesn't exist.

Prototypes

```
ASSERT_TRIGGER_EXISTS(ASCHEMA VARCHAR(128), ATRIGGER VARCHAR(128))
ASSERT_TRIGGER_EXISTS(ATRIGGER VARCHAR(128))
```

Description

Raises the ASSERT_FAILED_STATE state if the trigger specified by **ASCHEMA** and **ATRIGGER** does not exist. If not specified, **ASCHEMA** defaults to the value of the *CURRENT SCHEMA* special register.

Parameters

ASCHEMA Specifies the schema containing the trigger to check. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATRIGGER Specifies the name of the trigger to check.

Examples

Test the *UTILS.VACATIONS_INSERT* trigger exists:

```
CALL ASSERT_TRIGGER_EXISTS('UTILS', 'VACATIONS_INSERT');
```

Test the existence of the trigger *VACATIONS_DELETE* in the current schema:

```
CALL ASSERT_TRIGGER_EXISTS('VACATIONS_DELETE');
```

```
SQL0438N  Application raised error or warning with diagnostic text:
"DB2INST1.VACATIONS_DELETE                         does not exist".
SQLSTATE=90001
```

See Also

- Source code
- *ASSERT_COLUMN_EXISTS procedure*
- *ASSERT_TABLE_EXISTS procedure*
- *ASSERT_ROUTINE_EXISTS procedure*
- ASSERT_FAILED_STATE

AUTO_DELETE procedure

Automatically removes data from **DEST_TABLE** that doesn't exist in **SOURCE_TABLE**, based on **DEST_KEY**.

Prototypes

```
AUTO_DELETE(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_SCHEMA␣
↪VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_KEY VARCHAR(128))
AUTO_DELETE(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_SCHEMA␣
↪VARCHAR(128), DEST_TABLE VARCHAR(128))
AUTO_DELETE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_KEY VARCHAR(128))
AUTO_DELETE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128))
```

Description

The AUTO_DELETE procedure deletes rows from **DEST_TABLE** that do not exist in **SOURCE_TABLE**. This procedure is intended to be used after *AUTO_MERGE procedure* has been used to upsert from the source to the destination.

The **DEST_KEY** parameter specifies the name of the unique key to use for identifying rows in the destination table. If specified, it must be the name of a unique key or primary key which covers columns which exist in both the source and destination tables. If omitted, it defaults to the name of the primary key of the destination table.

If **SOURCE_SCHEMA** and **DEST_SCHEMA** are not specified they default to the current schema.

The destination table must have at least one unique key (or a primary key), and the executing user must have DELETE privileges on the destination table.

Parameters

SOURCE_SCHEMA If provided, specifies the schema containing **SOURCE_TABLE**. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE Specifies the name of the table within **SOURCE_SCHEMA** to read for the list of rows to be preserved.

DEST_SCHEMA If provided, specifies the schema containing **DEST_TABLE**. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_TABLE Specifies the name of the table within **DEST_SCHEMA** from which data will be deleted. This table *must* have at least one unique key (or a primary key).

DEST_KEY If provided, specifies the name of the unique key in the destination table which will be joined to the equivalently named fields in the source table to determine which rows to delete. If omitted, defaults to the name of the primary key of the destination table.

Examples

Merge new content from *EMP_SOURCE* into the *EMPLOYEES* table, matching rows via the primary key of *EMPLOYEES*, then delete rows in *EMPLOYEES* that no longer exist in *EMP_SOURCE:*

```
CALL AUTO_MERGE('EMP_SOURCE', 'EMPLOYEES');
CALL AUTO_DELETE('EMP_SOURCE', 'EMPLOYEES');
```

Delete content from *IW.CONTRACTS* that no longer exists in *STAGING.CONTRACTS*, using a specific unique key for matching rows:

```
CALL AUTO_DELETE('STAGING', 'CONTRACTS', 'IW', 'CONTRACTS', 'CONTRACTS_KEY');
```

See Also

- Source code
- *AUTO_MERGE procedure*
- *AUTO_INSERT procedure*

AUTO_INSERT procedure

Automatically inserts data into **DEST_TABLE** from **SOURCE_TABLE**.

Prototypes

```
AUTO_INSERT(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_SCHEMA␣
↪VARCHAR(128), DEST_TABLE VARCHAR(128))
AUTO_INSERT(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128))
```

Description

The AUTO_INSERT procedure inserts all data from **SOURCE_TABLE** into **DEST_TABLE** by means of an automatically generated INSERT statement covering all columns common to both tables.

If **SOURCE_SCHEMA** and **DEST_SCHEMA** are not specified they default to the current schema.

Only columns common to both the destination table and the source table will be included in the generated statement. Destination columns must be updateable (they cannot be defined as `GENERATED ALWAYS`), and the executing user must have INSERT privileges on the destination table.

Parameters

SOURCE_SCHEMA If provided, specifies the schema containing **SOURCE_TABLE**. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE Specifies the name of the table within **SOURCE_SCHEMA** from which to read data.

DEST_SCHEMA If provided, specifies the schema containing **DEST_TABLE**. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_TABLE Specifies the name of the table within **DEST_SCHEMA** into which data will be copied.

Examples

Insert all content from *NEW_EMP* into *EMPLOYEES*:

```
CALL AUTO_INSERT('NEW_EMP', 'EMPLOYEES');
```

Replace all content in *IW.CONTRACTS* with content from *STAGING.CONTRACTS*:

```
TRUNCATE IW.CONTRACTS
    REUSE STORAGE
    RESTRICT WHEN DELETE TRIGGERS
    IMMEDIATE;
CALL AUTO_INSERT('STAGING', 'CONTRACTS', 'IW', 'CONTRACTS');
```

See Also

- Source code
- *AUTO_MERGE procedure*

- *AUTO_DELETE procedure*

AUTO_MERGE procedure

Automatically inserts/updates ("upserts") data from **SOURCE_TABLE** into **DEST_TABLE**, based on **DEST_KEY**.

Prototypes

```
AUTO_MERGE(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_SCHEMA␣
↪VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_KEY VARCHAR(128))
AUTO_MERGE(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_SCHEMA␣
↪VARCHAR(128), DEST_TABLE VARCHAR(128))
AUTO_MERGE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_KEY VARCHAR(128))
AUTO_MERGE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128))
```

Description

The AUTO_MERGE procedure performs an "upsert", or combined insert and update of all data from **SOURCE_TABLE** into **DEST_TABLE** by means of an automatically generated MERGE statement.

The **DEST_KEY** parameter specifies the name of the unique key to use for identifying rows in the destination table. If specified, it must be the name of a unique key or primary key which covers columns which exist in both the source and destination tables. If omitted, it defaults to the name of the primary key of the destination table.

If **SOURCE_SCHEMA** and **DEST_SCHEMA** are not specified they default to the current schema.

Only columns common to both the destination table and the source table will be included in the generated statement. Destination columns must be updateable (they cannot be defined as GENERATED ALWAYS), and the executing user must have INSERT and UPDATE privileges on the destination table.

Parameters

SOURCE_SCHEMA If provided, specifies the schema containing **SOURCE_TABLE**. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE Specifies the name of the table within **SOURCE_SCHEMA** from which data will be read.

DEST_SCHEMA If provided, specifies the schema containing **DEST_TABLE**. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_TABLE Specifies the name of the table within **DEST_SCHEMA** into which data will be inserted or updated. This table *must* have at least one unique key (or a primary key).

DEST_KEY If provided, specifies the name of the unique key in the destination table which will be joined to the equivalently named fields in the source table to determine whether rows are to be inserted or updated. If omitted, defaults to the name of the primary key of the destination table.

Examples

Merge new content from *EMP_SOURCE* into the *EMPLOYEES* table, matching rows via the primary key of *EMPLOYEES*, then delete rows in *EMPLOYEES* that no longer exist in *EMP_SOURCE*:

```
CALL AUTO_MERGE('EMP_SOURCE', 'EMPLOYEES');
CALL AUTO_DELETE('EMP_SOURCE', 'EMPLOYEES');
```

Merge new content from *STAGING.CONTRACTS* into *IW.CONTRACTS*, using a specific unique key for matching rows:

```
CALL AUTO_MERGE('STAGING', 'CONTRACTS', 'IW', 'CONTRACTS', 'CONTRACTS_KEY');
```

See Also

- Source code
- *AUTO_DELETE procedure*
- *AUTO_INSERT procedure*

COPY_AUTH procedure

Grants all authorities held by the source to the target, provided they are not already held (i.e. does not "re-grant" authorities already held).

Prototypes

```
COPY_AUTH(SOURCE VARCHAR(128), SOURCE_TYPE VARCHAR(1), DEST VARCHAR(128), DEST_TYPE␣
↪VARCHAR(1), INCLUDE_PERSONAL VARCHAR(1))
COPY_AUTH(SOURCE VARCHAR(128), DEST VARCHAR(128), INCLUDE_PERSONAL VARCHAR(1))
COPY_AUTH(SOURCE VARCHAR(128), DEST VARCHAR(128))
```

Description

COPY_AUTH is a procedure which copies all authorizations from the source grantee (**SOURCE**) to the destination grantee (**DEST**). Note that the implementation does not preserve the grantor, although technically this would be possible by utilizing the SET SESSION USER facility introduced by DB2 9, nor does it remove extra permissions that the destination grantee already possessed prior to the call. Furthermore, method authorizations are not copied.

Parameters

SOURCE The name of the user, group, or role to copy permissions from.

SOURCE_TYPE One of `'U'`, `'G'`, or `'R'` indicating whether **SOURCE** refers to a user, group, or role respectively. If this parameter is omitted *AUTH_TYPE scalar function* will be used to determine the type of **SOURCE**.

DEST The name of the user, group, or role to copy permissions to.

DEST_TYPE One of `'U'`, `'G'`, or `'R'` indicating whether **DEST** refers to a user, group, or role respectively. If this parameter is omitted *AUTH_TYPE scalar function* will be used to determine the type of **DEST**.

INCLUDE_PERSONAL If this parameter is `'Y'` and **SOURCE** refers to a user, then permissions associated with the user's personal schema will be included in the transfer. Defaults to `'N'` if omitted.

Examples

Copy authorizations from the user *TOM* to the user *DICK*, excluding any permissions associated with the *TOM* schema.

```
CALL COPY_AUTH('TOM', 'DICK', 'N');
```

Copy permissions granted to a group called *FINANCE* to a role called *FINANCE* (the **INCLUDE_PERSONAL** parameter is set to `'N'` here, but is effectively redundant as **SOURCE_TYPE** is not `'U'`).

```
CALL COPY_AUTH('FINANCE', 'G', 'FINANCE', 'R', 'N');
```

See Also

- Source code
- *AUTH_TYPE scalar function*
- *AUTH_DIFF table function*
- *AUTHS_HELD table function*
- *MOVE_AUTH procedure*
- *REMOVE_AUTH procedure*

CREATE_EXCEPTION_TABLE procedure

Creates an exception table based on the structure of the specified table.

Prototypes

```
CREATE_EXCEPTION_TABLE(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_
↪SCHEMA VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_TBSPACE VARCHAR(18))
CREATE_EXCEPTION_TABLE(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_
↪SCHEMA VARCHAR(128), DEST_TABLE VARCHAR(128))
CREATE_EXCEPTION_TABLE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_
↪TBSPACE VARCHAR(18))
CREATE_EXCEPTION_TABLE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128))
CREATE_EXCEPTION_TABLE(SOURCE_TABLE VARCHAR(128))
```

Description

The CREATE_EXCEPTION_TABLE procedure creates, from a template table (specified by **SOURCE_SCHEMA** and **SOURCE_TABLE**), another table (named by **DEST_SCHEMA** and **DEST_TABLE**) designed to hold LOAD and SET INTEGRITY exceptions from the template table. The new table is identical to the template table, but contains two extra fields: *EXCEPT_MSG* (which stores information about the exception that occurred when loading or setting the integrity of the table), and *EXCEPT_TS*, a TIMESTAMP field indicating when the exception the occurred.

The **DEST_TBSPACE** parameter identifies the tablespace used to store the new table's data. If **DEST_TBSPACE** is omitted it defaults to the tablespace of the template table.

Of the other parameters, only **SOURCE_TABLE** is mandatory. If **DEST_TABLE** is not specified it defaults to the value of **SOURCE_TABLE** with a suffix of `'_EXCEPTIONS'`. If **SOURCE_SCHEMA** and **DEST_SCHEMA** are not specified they default to the value of the *CURRENT SCHEMA* special register.

Warning: If the specified table already exists, this procedure will replace it, potentially losing all its content. If the existing exceptions data is important to you, make sure you back it up before executing this procedure.

Note: All authorizations present on the source table will be copied to the destination table.

Parameters

SOURCE_SCHEMA If provided, specifies the schema containing the template table on which to base the design of the new exceptions table. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE Specifies the name of the template table within **SOURCE_SCHEMA**.

DEST_SCHEMA If provided, specifies the schema in which the new exceptions table will be created. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_TABLE If provided, specifies the name of the new exceptions table. If omitted, defaults to the value of **SOURCE_TABLE** with `'_EXCEPTIONS'` appended to it.

DEST_TBSPACE If provided, specifies the tablespace in which to store the physical data of the new exceptions table. Defaults to the tablespace containing the table specified by **SOURCE_SCHEMA** and **SOURCE_TABLE**.

Examples

Create a new exceptions table based on the design of the *FINANCE.LEDGER* table, called *EXCEPTIONS.LEDGER* in the *EXCEPTSPACE* tablespace, then load data into the source table, diverting exceptions to the new exceptions table:

```
CALL CREATE_EXCEPTION_TABLE('FINANCE', 'LEDGER', 'EXCEPTIONS', 'LEDGER', 'EXCEPTSPACE
↪');
LOAD FROM LEDGER.IXF OF IXF REPLACE INTO FINANCE.LEDGER
  FOR EXCEPTION EXCEPTIONS.LEDGER;
```

Create a new exceptions table based on the *EMPLOYEE* table in the current schema called *EMPLOYEE_EXCEPTIONS*, in the same tablespace as the source, then LOAD the source table, and finally run a SET INTEGRITY from the source to the new exceptions table:

```
CALL CREATE_EXCEPTION_TABLE('EMPLOYEE');
LOAD FROM EMPLOYEE.IXF OF IXF REPLACE INTO EMPLOYEE;
SET INTEGRITY FOR EMPLOYEE IMMEDIATE CHECKED
  FOR EXCEPTION IN EMPLOYEE USE EMPLOYEE_EXCEPTIONS;
```

See Also

- Source code
- *CREATE_EXCEPTION_VIEW procedure*
- LOAD (built-in command)
- SET INTEGRITY (built-in statement)
- Exception tables

CREATE_EXCEPTION_VIEW procedure

Creates a view based on the specified exception table which interprets the content of the *EXCEPT_MSG* column.

Prototypes

```
CREATE_EXCEPTION_VIEW(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_
↪SCHEMA VARCHAR(128), DEST_VIEW VARCHAR(128))
CREATE_EXCEPTION_VIEW(SOURCE_TABLE VARCHAR(128), DEST_VIEW VARCHAR(128))
CREATE_EXCEPTION_VIEW(SOURCE_TABLE VARCHAR(128))
```

Description

The CREATE_EXCEPTION_VIEW procedure creates a view on top of an exceptions table (presumably created with *CREATE_EXCEPTION_TABLE procedure*). The view uses a recursive common-table-expression to split the large *EXCEPT_MSG* field into several rows and several columns to allow for easier analysis. Instead of *EXCEPT_MSG*, the view contains the following exceptions-related fields:

EXCEPT_TYPE A CHAR(1) column containing one of the following values:

- `'K'` - check constraint violation
- `'F'` - foreign key violation
- `'G'` - generated column violation
- `'I'` - unique index violation
- `'L'` - datalink load violation
- `'D'` - cascaded deletion violation

EXCEPT_OBJECT A VARCHAR(n) column containing the fully qualified name of the object that caused the exception (e.g. the name of the check constraint, foreign key, column or unique index)

Like *CREATE_EXCEPTION_TABLE procedure*, this procedure has only one mandatory parameter: **SOURCE_TABLE**. If **SOURCE_SCHEMA** and **DEST_SCHEMA** are not specified, they default to the value of the *CURRENT SCHEMA* special register. If **DEST_VIEW** is not specified, it defaults to the value of **SOURCE_TABLE** with a `'_V'` suffix.

Note: SELECT and CONTROL authorizations are copied from the source table to the destination view (INSERT, UPDATE, and DELETE authorizations are ignored).

Parameters

SOURCE_SCHEMA If provided, the schema containing the exception table on which to base the new view. Defaults to the value of the *CURRENT SCHEMA* special register if omitted.

SOURCE_TABLE Specifies the exception table on which to base the new view. This table is expected to have two columns named *EXCEPT_TS* and *EXCEPT_MSG*.

DEST_SCHEMA If provided, the schema in which to create the new view. Defaults to the value of the *CURRENT SCHEMA* special register if omitted.

DEST_VIEW If provided, the name of the new view. Defaults to **SOURCE_TABLE** with a `'_V'` suffix if omitted.

Examples

Create a view to interpret the content of *EXCEPTIONS.LEDGER* called *FINANCE.LEDGER_EXCEPTIONS*:

```
CALL CREATE_EXCEPTION_VIEW('EXCEPTIONS', 'LEDGER', 'FINANCE', 'LEDGER_EXCEPTIONS');
```

Create a view called *EMPLOYEE_EXCEPTIONS_V* based on the *EMPLOYEE_EXCEPTIONS* table in the current schema:

```
CALL CREATE_EXCEPTION_VIEW('EMPLOYEE_EXCEPTIONS');
```

See Also

- Source code
- *CREATE_EXCEPTION_TABLE procedure*
- Exception tables

CREATE_HISTORY_CHANGES procedure

Creates an "OLD vs NEW" changes view on top of the specified history table.

Prototypes

```
CREATE_HISTORY_CHANGES(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_
↪SCHEMA VARCHAR(128), DEST_VIEW VARCHAR(128))
CREATE_HISTORY_CHANGES(SOURCE_TABLE VARCHAR(128), DEST_VIEW VARCHAR(128))
CREATE_HISTORY_CHANGES(SOURCE_TABLE VARCHAR(128))
```

Description

The CREATE_HISTORY_CHANGES procedure creates a view on top of a history table which is assumed to have a structure generated by *CREATE_HISTORY_TABLE procedure*. The view represents the history data as a series of "change" rows. The *EFFECTIVE* and *EXPIRY* columns from the source history table are merged into a *CHANGED* column, a *CHANGE* column is calculated to show whether each change was an insertion, update, or deletion, and all other columns are represented twice as *OLD_* and *NEW_* variants.

If **DEST_VIEW** is not specified it defaults to the value of **SOURCE_TABLE** with `'_HISTORY'` replaced with `'_CHANGES'`. If **DEST_SCHEMA** and **SOURCE_SCHEMA** are not specified they default to the current schema.

Note: All SELECT and CONTROL authorities present on the source table will be copied to the destination table.

The type of change can be determined by querying the CHANGE column in the new view. The possible values (and their criteria) are:

CHANGE value	Criteria
`'INSERT'`	If the old key or keys are NULL and the new are non-NULL,the change was an insertion.
`'UPDATE'`	If both the old and new key or keys are non-NULL the change was an update.
`'DELETE'`	If the old key or keys are non-NULL and the new are NULL the change was a deletion.
`'ERROR'`	This should never happen!

Parameters

SOURCE_SCHEMA If provided, specifies the schema containing the history table on which to base the new changes view. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE The name of the history table on which to base the new changes view.

DEST_SCHEMA If provided, specifies the schema which will contain the new changes view. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_VIEW If provided, specifies the name of the new changes view. If omitted, defaults to **SOURCE_TABLE** with `'_HISTORY'` replaced with `'_CHANGES'`.

Examples

Create a *CUSTOMERS* table in the current schema, then create a history table called *CUSTOMERS_HISTORY* based upon on the *CUSTOMERS* table with DAY resolution. Install the triggers which will keep the history table up to date with the base table, and finally create a view that will provide old vs. new comparisons of the history:

```sql
CREATE TABLE CUSTOMERS (
  ID         INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  NAME       VARCHAR(100) NOT NULL,
  ADDRESS    VARCHAR(2000) NOT NULL,
  SECTOR     CHAR(2) NOT NULL
) COMPRESS YES;
CALL CREATE_HISTORY_TABLE('CUSTOMERS', 'DAY');
CALL CREATE_HISTORY_TRIGGERS('CUSTOMERS', 'DAY');
CALL CREATE_HISTORY_CHANGES('CUSTOMERS_HISTORY');
```

The structure of the resulting tables and views can be seen below:

```
$ db2 DESCRIBE TABLE CUSTOMERS

                                Data type                     Column
Column name                     schema    Data type name      Length     Scale Nulls
------------------------------- --------- ------------------- ---------- ----- ------
ID                              SYSIBM    INTEGER                      4     0 No
NAME                            SYSIBM    VARCHAR                    100     0 No
ADDRESS                         SYSIBM    VARCHAR                   2000     0 No
SECTOR                          SYSIBM    CHARACTER                    2     0 No

  4 record(s) selected.


$ db2 DESCRIBE TABLE CUSTOMERS_HISTORY

                                Data type                     Column
Column name                     schema    Data type name      Length     Scale Nulls
------------------------------- --------- ------------------- ---------- ----- ------
```

```
EFFECTIVE_DAY                    SYSIBM    DATE                           4     0 No
EXPIRY_DAY                       SYSIBM    DATE                           4     0 No
ID                               SYSIBM    INTEGER                        4     0 No
NAME                             SYSIBM    VARCHAR                      100     0 No
ADDRESS                          SYSIBM    VARCHAR                     2000     0 No
SECTOR                           SYSIBM    CHARACTER                      2     0 No

  6 record(s) selected.


$ db2 DESCRIBE TABLE CUSTOMERS_CHANGES

                                 Data type                     Column
Column name                      schema    Data type name      Length     Scale Nulls
-------------------------------- --------- ------------------- ---------- ----- ------
CHANGED                          SYSIBM    DATE                           4     0 Yes
CHANGE                           SYSIBM    CHARACTER                      6     0 No
OLD_ID                           SYSIBM    INTEGER                        4     0 Yes
NEW_ID                           SYSIBM    INTEGER                        4     0 Yes
OLD_NAME                         SYSIBM    VARCHAR                      100     0 Yes
NEW_NAME                         SYSIBM    VARCHAR                      100     0 Yes
OLD_ADDRESS                      SYSIBM    VARCHAR                     2000     0 Yes
NEW_ADDRESS                      SYSIBM    VARCHAR                     2000     0 Yes
OLD_SECTOR                       SYSIBM    CHARACTER                      2     0 Yes
NEW_SECTOR                       SYSIBM    CHARACTER                      2     0 Yes

  10 record(s) selected.
```

See Also

- Source code
- *CREATE_HISTORY_TABLE procedure*
- *CREATE_HISTORY_SNAPSHOTS procedure*
- *CREATE_HISTORY_TRIGGERS procedure*
- History design usenet post
- CREATE TABLE (built-in command)
- CREATE VIEW (built-in command)
- Time Travel Queries in DB2 v10.1

CREATE_HISTORY_SNAPSHOTS procedure

Creates an exploded view of the specified history table with one row per entity per resolution time-slice (e.g. daily, monthly, yearly, etc.)

Prototypes

```
CREATE_HISTORY_SNAPSHOTS(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_
↪SCHEMA VARCHAR(128), DEST_VIEW VARCHAR(128), RESOLUTION VARCHAR(11))
CREATE_HISTORY_SNAPSHOTS(SOURCE_TABLE VARCHAR(128), DEST_VIEW VARCHAR(128),␣
↪RESOLUTION VARCHAR(11))
CREATE_HISTORY_SNAPSHOTS(SOURCE_TABLE VARCHAR(128), RESOLUTION VARCHAR(11))
```

Description

The CREATE_HISTORY_SNAPSHOTS procedure creates a view on top of a history table which is assumed to have a structure generated by *CREATE_HISTORY_TABLE procedure*. The view represents the history data as a series of "snapshots" of the main table at various points through time. The *EFFECTIVE* and *EXPIRY* columns from the source history table are replaced with a *SNAPSHOT* column which indicates the timestamp or date of the snapshot of the main table. All other columns are represented in their original form.

If **DEST_VIEW** is not specified it defaults to the value of **SOURCE_TABLE** with `'_HISTORY'` replaced with a custom suffix which depends on the value of **RESOLUTION**. For example, if **RESOLUTION** is `'MONTH'` then the suffix is `'MONTHLY'`, if **RESOLUTION** is `'WEEK'`, or `'WEEK_ISO'` then the suffix is `'WEEKLY'` and so on. If **DEST_SCHEMA** and **SOURCE_SCHEMA** are not specified they default to the current schema.

The **RESOLUTION** parameter determines the amount of time between snapshots. Snapshots will be generated for the end of each period given by a particular **RESOLUTION**. For example, if **RESOLUTION** is `'WEEK'` then a snapshot will be generated for the end of each week according to *WEEKEND scalar function* from the earliest record in the history table up to the current date. See *CREATE_HISTORY_TRIGGERS procedure* for a list of the possible values.

Note: All SELECT and CONTROL authorities present on the source table will be copied to the destination table.

Parameters

SOURCE_SCHEMA If provided, specifies the schema containing the history table on which to base the new changes view. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE The name of the history table on which to base the new snapshots view.

DEST_SCHEMA If provided, specifies the schema which will contain the new snapshots view. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_VIEW If provided, specifies the name of the new snapshots view. If omitted, defaults to **SOURCE_TABLE** with `'_HISTORY'` replaced with a suffix determined by the **RESOLUTION** parameter.

RESOLUTION Specifies the smallest unit of time that an entry in the view can cover. See *CREATE_HISTORY_TRIGGERS procedure* for a list of possible values. This should be greater than or equal to the **RESOLUTION** specified when the source table was created with *CREATE_HISTORY_TABLE procedure* (it is nonsensical to create a snapshot at finer resolution).

Examples

Create an *INVOICES* table in the current schema, then create a history table called *INVOICES_HISTORY* based on the *INVOICES* table with DAY resolution. Install the triggers which will keep the history table up to date with the base table, and finally create a view that will provide a weekly snapshot of the data:

```
CREATE TABLE INVOICES (
  INVOICE     CHAR(8) NOT NULL PRIMARY KEY,
  CUSTOMER    CHAR(8) NOT NULL REFERENCES CUSTOMERS(CUSTOMER),
  ORDER       INTEGER NOT NULL REFERENCES ORDERS(ORDER),
  AMOUNT      DECIMAL(17,2) NOT NULL,
  PAID        DATE DEFAULT NULL
) COMPRESS YES;
CALL CREATE_HISTORY_TABLE('INVOICES', 'DAY');
CALL CREATE_HISTORY_TRIGGERS('INVOICES', 'DAY');
CALL CREATE_HISTORY_SNAPSHOTS('INVOICES_HISTORY', 'WEEK');
```

The structure of the resulting tables and views can be seen below:

```
$ db2 DESCRIBE TABLE INVOICES

                                Data type                     Column
Column name                     schema    Data type name      Length     Scale Nulls
------------------------------- --------- ------------------- ---------- ----- ------
INVOICE                         SYSIBM    CHARACTER                    8     0 No
CUSTOMER                        SYSIBM    CHARACTER                    8     0 No
ORDER                           SYSIBM    INTEGER                      4     0 No
AMOUNT                          SYSIBM    DECIMAL                     17     2 No
PAID                            SYSIBM    DATE                         4     0 Yes

  5 record(s) selected.

$ db2 DESCRIBE TABLE INVOICES_HISTORY

                                Data type                     Column
Column name                     schema    Data type name      Length     Scale Nulls
------------------------------- --------- ------------------- ---------- ----- ------
EFFECTIVE_DAY                   SYSIBM    DATE                         4     0 No
EXPIRY_DAY                      SYSIBM    DATE                         4     0 No
INVOICE                         SYSIBM    CHARACTER                    8     0 No
CUSTOMER                        SYSIBM    CHARACTER                    8     0 No
ORDER                           SYSIBM    INTEGER                      4     0 No
AMOUNT                          SYSIBM    DECIMAL                     17     2 No
PAID                            SYSIBM    DATE                         4     0 Yes

  7 record(s) selected.

$ db2 DESCRIBE TABLE INVOICES_WEEKLY

                                Data type                     Column
Column name                     schema    Data type name      Length     Scale Nulls
------------------------------- --------- ------------------- ---------- ----- ------
SNAPSHOT                        SYSIBM    DATE                         4     0 Yes
INVOICE                         SYSIBM    CHARACTER                    8     0 No
CUSTOMER                        SYSIBM    CHARACTER                    8     0 No
ORDER                           SYSIBM    INTEGER                      4     0 No
AMOUNT                          SYSIBM    DECIMAL                     17     2 No
PAID                            SYSIBM    DATE                         4     0 Yes

  6 record(s) selected.
```

See Also

- Source code
- *CREATE_HISTORY_TABLE procedure*
- *CREATE_HISTORY_CHANGES procedure*
- *CREATE_HISTORY_TRIGGERS procedure*
- History design usenet post
- CREATE TABLE (built-in command)
- CREATE VIEW (built-in command)
- Time Travel Queries in DB2 v10.1

CREATE_HISTORY_TABLE procedure

Creates a temporal history table based on the structure of the specified table.

Prototypes

```
CREATE_HISTORY_TABLE(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_
↪SCHEMA VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_TBSPACE VARCHAR(18), RESOLUTION␣
↪VARCHAR(11))
CREATE_HISTORY_TABLE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128), DEST_TBSPACE␣
↪VARCHAR(18), RESOLUTION VARCHAR(11))
CREATE_HISTORY_TABLE(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128), RESOLUTION␣
↪VARCHAR(11))
CREATE_HISTORY_TABLE(SOURCE_TABLE VARCHAR(128), RESOLUTION VARCHAR(11))
```

Description

The CREATE_HISTORY_TABLE procedure creates, from a template table specified by **SOURCE_SCHEMA** and **SOURCE_TABLE**, another table named by **DEST_SCHEMA** and **DEST_TABLE** designed to hold a representation of the source table's content over time. Specifically, the destination table has the same structure as source table, but with two additional columns named *EFFECTIVE_time_period* and *EXPIRY_time_period* (where *time_period* is determined by the **RESOLUTION** parameter), which occur before all other "original" columns. The primary key of the source table, in combination with *EFFECTIVE_time_period* will form the primary key of the destination table, and a unique index involving the primary key and the *EXPIRY_time_period* column will also be created as this provides better performance of the triggers used to maintain the destination table.

The **DEST_TBSPACE** parameter identifies the tablespace used to store the new table's data. If **DEST_TBSPACE** is not specified, it defaults to the tablespace of the source table. If **DEST_TABLE** is not specified it defaults to the value of **SOURCE_TABLE** with `'_HISTORY'` as a suffix. If **DEST_SCHEMA** and **SOURCE_SCHEMA** are not specified they default to the current schema.

The **RESOLUTION** parameter determines the smallest unit of time that a history record can cover. See *CREATE_HISTORY_TRIGGERS procedure* for a list of the possible values.

All SELECT and CONTROL authorities present on the source table will be copied to the destination table. However, INSERT, UPDATE and DELETE authorities are excluded as these operations should only ever be performed by the history maintenance triggers themselves. The compression status of the source table will be copied to the destination table.

Warning: If the specified table already exists, this procedure will replace it, potentially losing all its content. If the existing history data is important to you, make sure you back it up before executing this procedure.

Note: This procedure is mostly redundant as of DB2 v10.1 which includes the ability to create temporal tables automatically via the `PERIOD` element combined with `SYSTEM TIME` and `BUSINESS TIME` specifications. However, the DB2 v10.1 implementation does not include the ability to create temporal tables with particularly coarse resolutions like `WEEK`.

Parameters

SOURCE_SCHEMA If provided, specifies the schema containing the template table on which to base the design of the new history table. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE Specifies the name of the template table within **SOURCE_SCHEMA**.

DEST_SCHEMA If provided, specifies the schema in which the new exceptions table will be created. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_TABLE If provided, specifies the name of the new exceptions table. If omitted, defaults to the value of **SOURCE_TABLE** with `'_HISTORY'` appended to it.

DEST_TBSPACE The name of the tablespace in which the history table should be created. If omitted, defaults to the tablespace in which **SOURCE_TABLE** exists.

RESOLUTION Specifies the granularity of the history to be stored. See *CREATE_HISTORY_TRIGGERS procedure* for a description of the possible values.

Examples

Create a *CORP.CUSTOMERS* table, then create a history table called *CORP.CUSTOMERS_HISTORY* based upon on the *CORP.CUSTOMERS* table in the *CORPSPACE* tablespace with DATE resolution. Finally, install the triggers which will keep the history table up to date with the base table:

```
CREATE TABLE CORP.CUSTOMERS (
  ID          INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  NAME        VARCHAR(100) NOT NULL,
  ADDRESS     VARCHAR(2000) NOT NULL,
  SECTOR      CHAR(2) NOT NULL REFERENCES SECTORS(SECTOR)
) IN CORPSPACE COMPRESS YES;
CALL CREATE_HISTORY_TABLE('CORP', 'CUSTOMERS', 'CORP', 'CUSTOMERS_HISTORY', 'CORPSPACE
↪', 'DAY');
CALL CREATE_HISTORY_TRIGGERS('CORP', 'CUSTOMERS', 'CORP', 'CUSTOMERS_HISTORY', 'DAY',
↪'');
```

The same example as above, but eliminating as many optional parameters as possible:

```
SET SCHEMA CORP;
CREATE TABLE CUSTOMERS (
  ID          INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  NAME        VARCHAR(100) NOT NULL,
  ADDRESS     VARCHAR(2000) NOT NULL,
  SECTOR      CHAR(2) NOT NULL REFERENCES SECTORS(SECTOR),
) IN CORPSPACE COMPRESS YES;
```

```
CALL CREATE_HISTORY_TABLE('CUSTOMERS', 'DAY');
CALL CREATE_HISTORY_TRIGGERS('CUSTOMERS', 'DAY');
```

Create a history table on top of an existing populated customers table called *CORP.CUSTOMERS*. Note that before creating the triggers that link the base table to the history table, we insert the existing rows from *CORP.CUSTOMERS* into *CORP.CUSTOMERS_HISTORY* with some appropriate effective and expiry values (in future a procedure may be provided to perform this step automatically):

```
SET SCHEMA CORP;
CALL CREATE_HISTORY_TABLE('CUSTOMERS', 'DAY');
INSERT INTO CUSTOMERS_HISTORY SELECT CURRENT DATE, '9999-12-31', T.* FROM CUSTOMERS T;
CALL CREATE_HISTORY_TRIGGERS('CUSTOMERS', 'DAY');
```

See Also

- Source code
- *CREATE_HISTORY_TRIGGERS procedure*
- *CREATE_HISTORY_CHANGES procedure*
- *CREATE_HISTORY_SNAPSHOTS procedure*
- History design usenet post
- CREATE TABLE (built-in command)
- Time Travel Queries in DB2 v10.1

CREATE_HISTORY_TRIGGERS procedure

Creates the triggers to link the specified table to its corresponding history table.

Prototypes

```
CREATE_HISTORY_TRIGGERS(SOURCE_SCHEMA VARCHAR(128), SOURCE_TABLE VARCHAR(128), DEST_
↪SCHEMA VARCHAR(128), DEST_TABLE VARCHAR(128), RESOLUTION VARCHAR(11), OFFSET␣
↪VARCHAR(100))
CREATE_HISTORY_TRIGGERS(SOURCE_TABLE VARCHAR(128), DEST_TABLE VARCHAR(128),␣
↪RESOLUTION VARCHAR(11), OFFSET VARCHAR(100))
CREATE_HISTORY_TRIGGERS(SOURCE_TABLE VARCHAR(128), RESOLUTION VARCHAR(11), OFFSET␣
↪VARCHAR(100))
CREATE_HISTORY_TRIGGERS(SOURCE_TABLE VARCHAR(128), RESOLUTION VARCHAR(11))
```

Description

The CREATE_HISTORY_TRIGGERS procedure creates several trigger linking the specified source table to the destination table which is assumed to have a structure compatible with the result of running *CREATE_HISTORY_TABLE procedure*, i.e. two extra columns called *EFFECTIVE_time_period* and *EXPIRY_time_period*.

If **DEST_TABLE** is not specified it defaults to the value of **SOURCE_TABLE** with `'_HISTORY'` as a suffix. If **DEST_SCHEMA** and **SOURCE_SCHEMA** are not specified they default to the current schema.

The **RESOLUTION** parameter specifies the smallest unit of time that a history entry can cover. This is effectively used to quantize the history. The value given for the **RESOLUTION** parameter should match the value given as the **RESOLUTION** parameter to *CREATE_HISTORY_TABLE procedure*. The values which can be specified are as follows:

Value	Meaning
`'MICROSECOND'`	With this value, the triggers perform no explicit quantization. Instead, history records are constrained simply by the resolution of the TIMESTAMP datatype, currently microseconds.
`'SECOND'`	Quantizes history into individual seconds. If multiple changes occur to the master record within a single second, only the final state is kept in the history table.
`'MINUTE'`	Quantizes history into individual minutes.
`HOUR'`	Quantizes history into individual hours.
`'DAY'`	Quantizes history into individual days. If multiple changes occur to the master record within a single day, as defined by the CURRENT DATE special register, only the final state is kept in the history table.
`'WEEK'`	Quantizes history into blocks starting on a Sunday and ending on a Saturday.
`'WEEK_ISO'`	Quantizes history into blocks starting on a Monday and ending on a Sunday.
`'MONTH'`	Quantizes history into blocks starting on the 1st of a month and ending on the last day of the corresponding month.
`'YEAR'`	Quantizes history into blocks starting on the 1st of a year and ending on the last day of the corresponding year.

The **OFFSET** parameter specifies an SQL phrase that will be used to offset the effective dates of new history records. For example, if the source table is only updated a week in arrears, then **OFFSET** could be set to `'- 7 DAYS'` to cause the effective dates to be accurate. If offset is not specified a blank string `''` (meaning no offset) is used.

Note: This procedure is mostly redundant as of DB2 v10.1 which includes the ability to create temporal tables automatically via the `PERIOD` element combined with `SYSTEM TIME` and `BUSINESS TIME` specifications. However, the DB2 v10.1 implementation does not include the ability to create temporal tables with particularly coarse resolutions like `WEEKLY`.

Parameters

SOURCE_SCHEMA If provided, the schema of the table on which to define the triggers. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

SOURCE_TABLE The name of the table on which to define the triggers.

DEST_SCHEMA If provided, the schema of the table which the triggers should write rows to. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

DEST_TABLE If provided, the name of the table which the triggers should write rows into. If omitted, defaults to the value of the **SOURCE_TABLE** parameter with `'_HISTORY'` appended.

RESOLUTION The time period to which the triggers should quantize the history records. Should be the same as the resolution specified when creating the history table with *CREATE_HISTORY_TABLE procedure*.

OFFSET A string specifying an offset (in the form of a labelled duration) which will be applied to the effective dates written by the triggers. If omitted, defaults to the empty string `''` (meaning no offset is to be applied).

Examples

Create a *CORP.CUSTOMERS* table, then create a history table called *CORP.CUSTOMERS_HISTORY* based upon on the *CORP.CUSTOMERS* table in the *CORPSPACE* tablespace with DATE resolution. Finally, install the triggers which will keep the history table up to date with the base table:

```
CREATE TABLE CORP.CUSTOMERS (
  ID          INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  NAME        VARCHAR(100) NOT NULL,
  ADDRESS     VARCHAR(2000) NOT NULL,
  SECTOR      CHAR(2) NOT NULL REFERENCES SECTORS(SECTOR)
) IN CORPSPACE COMPRESS YES;
CALL CREATE_HISTORY_TABLE('CORP', 'CUSTOMERS', 'CORP', 'CUSTOMERS_HISTORY', 'CORPSPACE
↪', 'DAY');
CALL CREATE_HISTORY_TRIGGERS('CORP', 'CUSTOMERS', 'CORP', 'CUSTOMERS_HISTORY', 'DAY',
↪'');
```

Create a history table for an existing *PROJECTS* table. Populate it with the existing data (and appropriate effective and expiry dates), then create the history triggers to link the *PROJECTS* table to the *PROJECTS_HISTORY* table, with a weekly resolution and a 1 week history offset:

```
CALL CREATE_HISTORY_TABLE('PROJECTS', 'WEEK');
INSERT INTO PROJECTS_HISTORY SELECT WEEKSTART(CURRENT DATE), DATE('9999-12-31'), T.*␣
↪FROM PROJECTS T;
CALL CREATE_HISTORY_TRIGGERS('PROJECTS_HISTORY', 'WEEK', '- 7 DAYS');
```

See Also

- Source code
- *CREATE_HISTORY_TABLE procedure*
- *CREATE_HISTORY_CHANGES procedure*
- *CREATE_HISTORY_SNAPSHOTS procedure*
- History design usenet post
- CREATE TABLE (built-in command)
- CREATE TRIGGER (built-in command)
- Time Travel Queries in DB2 v10.1

DISABLE_TRIGGER procedure

Disables the specified trigger by saving its definition to a table and dropping it.

Prototypes

```
DISABLE_TRIGGER(ASCHEMA VARCHAR(128), ATRIGGER VARCHAR(128))
DISABLE_TRIGGER(ATRIGGER VARCHAR(128))
```

Description

Drops a trigger after storing its definition in DISABLED_TRIGGERS for later "revival" with *ENABLE_TRIGGER procedure*. The trigger must be operative (if it is not, recreate it with *RECREATE_TRIGGER procedure* before calling DISABLE_TRIGGER.

Parameters

ASCHEMA If provided, the schema containing the trigger to disable. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATRIGGER The name of the trigger to disable.

Examples

Disable the *FINANCE.LEDGER_INSERT* trigger:

```
CALL DISABLE_TRIGGER('FINANCE', 'LEDGER_INSERT');
```

Recreate then disable the *EMPLOYEE_UPDATE* trigger in the current schema:

```
CALL RECREATE_TRIGGER('EMPLOYEE_UPDATE');
CALL DISABLE_TRIGGER('EMPLOYEE_UPDATE');
```

See Also

- Source code
- *ENABLE_TRIGGER procedure*
- *RECREATE_TRIGGER procedure*
- *DISABLE_TRIGGERS procedure*
- SYSCAT.TRIGGERS (built-in catalogue table)

DISABLE_TRIGGERS procedure

Disables all triggers associated with the specified table by saving their definitions to a table and dropping them.

Prototypes

```
DISABLE_TRIGGERS(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128))
DISABLE_TRIGGERS(ATABLE VARCHAR(128))
```

Description

Disables all the operative triggers associated with a particular table. If a trigger exists, but is marked inoperative, it is not touched by this procedure. You can recreate such triggers with *RECREATE_TRIGGER procedure* before calling DISABLE_TRIGGERS.

Parameters

ASCHEMA If provided, the schema containing the table for which to disable triggers. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATABLE The name of the table to disable all active triggers for.

Examples

Disable all triggers on the *FINANCE.LEDGER* table:

```
CALL DISABLE_TRIGGERS('FINANCE', 'LEDGER');
```

Disable the triggers for the *EMPLOYEE* table in the current schema:

```
CALL DISABLE_TRIGGERS('EMPLOYEE');
```

See Also

- Source code
- *ENABLE_TRIGGERS procedure*
- *RECREATE_TRIGGER procedure*
- *DISABLE_TRIGGER procedure*
- SYSCAT.TRIGGERS (built-in catalogue table)

DROP_SCHEMA procedure

Drops **ASCHEMA** and all objects within it.

Prototypes

```
DROP_SCHEMA(ASCHEMA VARCHAR(128))
```

Description

DROP_SCHEMA is a utility procedure which drops all objects (tables, views, triggers, sequences, aliases, etc.) in a schema and then drops the schema. It was originally intended to make destruction of user-owned schemas easier (in the event that a user no longer required access) but can also be used to make writing upgrade scripts easier.

Note: This procedure is effectively redundant since DB2 9.5 which includes the built-in procedure ADMIN_DROP_SCHEMA, albeit with a somewhat more complicated calling convention.

Parameters

ASCHEMA The name of the schema to drop.

Examples

Drop the *FRED* schema and all objects within it:

```
CALL DROP_SCHEMA('FRED');
```

Drop all schemas which start with the characters *TEST:*

```
BEGIN ATOMIC
  FOR T AS
    SELECT SCHEMANAME
    FROM SYSCAT.SCHEMATA
    WHERE SCHEMANAME LIKE 'TEST%'
  DO
    CALL DROP_SCHEMA(T.SCHEMANAME);
  END FOR;
END!
```

See Also

- Source code
- ADMIN_DROP_SCHEMA (built-in procedure)

ENABLE_TRIGGER procedure

Enables the specified trigger by restoring its definition from a table.

Prototypes

```
ENABLE_TRIGGER(ASCHEMA VARCHAR(128), ATRIGGER VARCHAR(128))
ENABLE_TRIGGER(ATRIGGER VARCHAR(128))
```

Description

Restores a previously disabled trigger by reading its definition from DISABLED_TRIGGERS and recreating it. The trigger must have been disabled with *DISABLE_TRIGGER procedure* or *DISABLE_TRIGGERS procedure*.

Parameters

ASCHEMA If provided, the schema containing the trigger to enable. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATRIGGER The name of the trigger to enable.

Examples

Enable the *FINANCE.LEDGER_INSERT* trigger:

```
CALL ENABLE_TRIGGER('FINANCE', 'LEDGER_INSERT');
```

Enable the *EMPLOYEE_UPDATE* trigger in the current schema:

```
CALL ENABLE_TRIGGER('EMPLOYEE_UPDATE');
```

See Also

- Source code
- *DISABLE_TRIGGER procedure*
- *ENABLE_TRIGGERS procedure*
- SYSCAT.TRIGGERS (built-in catalogue table)

ENABLE_TRIGGERS procedure

Enables all triggers associated with a specified table.

Prototypes

```
ENABLE_TRIGGERS(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128))
ENABLE_TRIGGERS(ATABLE VARCHAR(128))
```

Description

Enables all the disabled triggers associated with a particular table. Note that this does not affect inactive triggers which are still attached to the table, just those triggers that have been disabled with *DISABLE_TRIGGER procedure* or *DISABLE_TRIGGERS procedure*. To reactivate inactive triggers, see *RECREATE_TRIGGER procedure* and *RECREATE_TRIGGERS procedure*.

Parameters

ASCHEMA If provided, the schema containing the table for which to enable triggers. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

ATABLE The name of the table to enable all disabled triggers for.

Examples

Enable all disabled triggers on the *FINANCE.LEDGER* table:

```
CALL ENABLE_TRIGGERS('FINANCE', 'LEDGER');
```

Enable the disabled triggers for the *EMPLOYEE* table in the current schema:

```
CALL ENABLE_TRIGGERS('EMPLOYEE');
```

See Also

- Source code
- *DISABLE_TRIGGERS procedure*
- *RECREATE_TRIGGER procedure*
- *ENABLE_TRIGGER procedure*
- SYSCAT.TRIGGERS (built-in catalogue table)

MOVE_AUTH procedure

Moves all authorities held by the source to the target, provided they are not already held.

Prototypes

```
MOVE_AUTH(SOURCE VARCHAR(128), SOURCE_TYPE VARCHAR(1), DEST VARCHAR(128), DEST_TYPE␣
↪VARCHAR(1), INCLUDE_PERSONAL VARCHAR(1))
MOVE_AUTH(SOURCE VARCHAR(128), DEST VARCHAR(128), INCLUDE_PERSONAL VARCHAR(1))
MOVE_AUTH(SOURCE VARCHAR(128), DEST VARCHAR(128))
```

Description

MOVE_AUTH is a procedure which moves all authorizations from the source grantee (**SOURCE**) to the destination grantee (**DEST**). Like *COPY_AUTH procedure*, this procedure does not preserve the grantor, and method authorizations are not moved. Essentially this procedure combines *COPY_AUTH procedure* and *REMOVE_AUTH procedure* to copy authorizations from **SOURCE** to **DEST** and then remove them from **SOURCE**.

Note: Column-level authorizations will be copied to **DEST**, but will not be removed from **SOURCE**. Their removal must be handled manually.

Parameters

SOURCE The name of the user, group, or role to copy permissions from.

SOURCE_TYPE One of `'U'`, `'G'`, or `'R'` indicating whether **SOURCE** refers to a user, group, or role respectively. If this parameter is omitted *AUTH_TYPE scalar function* will be used to determine the type of **SOURCE**.

DEST The name of the user, group, or role to copy permissions to.

DEST_TYPE One of `'U'`, `'G'`, or `'R'` indicating whether **DEST** refers to a user, group, or role respectively. If this parameter is omitted *AUTH_TYPE scalar function* will be used to determine the type of **DEST**.

INCLUDE_PERSONAL If this parameter is `'Y'` and **SOURCE** refers to a user, then permissions associated with the user's personal schema will be included in the transfer. Defaults to `'N'` if omitted.

Examples

Copy authorizations from the user *TOM* to the user *DICK*, excluding any permissions associated with the *TOM* schema (so *TOM* retains access to his personal schema after this command).

```
CALL MOVE_AUTH('TOM', 'DICK', 'N');
```

Move permissions granted to a group called *FINANCE* to a role called *FINANCE* (the **INCLUDE_PERSONAL** parameter is set to `'N'` here, but is effectively redundant as **SOURCE_TYPE** is not `'U'`).

```
CALL MOVE_AUTH('FINANCE', 'G', 'FINANCE', 'R', 'N');
```

See Also

- Source code
- *AUTH_TYPE scalar function*
- *AUTH_DIFF table function*
- *AUTHS_HELD table function*
- *COPY_AUTH procedure*
- *REMOVE_AUTH procedure*

RECREATE_TRIGGER procedure

Recreates the specified inoperative trigger from its definition in the system catalogue.

Prototypes

```
RECREATE_TRIGGER(ASCHEMA VARCHAR(128), ATRIGGER VARCHAR(128))
RECREATE_TRIGGER(ATRIGGER VARCHAR(128))
```

Description

RECREATE_TRIGGER is a utility procedure which recreates the specified trigger using the SQL found in the system catalogue tables. It is useful for quickly recreating triggers which have been marked inoperative after a change to one or more of the trigger's dependencies. If **ASCHEMA** is omitted it defaults to the current schema.

Warning: The procedure does *not* drop the trigger before recreating it. This guards against attempting to recreate an operative trigger (an inoperative trigger can be recreated without dropping it first). That said, it will not return an error in the case of attempting to recreate an operative trigger; the procedure will simply do nothing.

Parameters

ASCHEMA If provided, the schema containing the trigger to recreate. If omitted, this parameter defaults to the value of the *CURRENT SCHEMA* special register.

ATRIGGER The name of the trigger to recreate.

Examples

Recreate the *FINANCE.LEDGER_INSERT* trigger:

```
CALL RECREATE_TRIGGER('FINANCE', 'LEDGER_INSERT');
```

Recreate the *EMPLOYEE_UPDATE* trigger in the current schema:

```
CALL RECREATE_TRIGGER('EMPLOYEE_UPDATE');
```

See Also

- Source code
- *RECREATE_TRIGGERS procedure*
- SYSCAT.TRIGGERS (buit-in catalogue view)

RECREATE_TRIGGERS procedure

Recreates all the inoperative triggers associated with the specified table from their definitions in the system catalogue.

Prototypes

```
RECREATE_TRIGGERS(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128))
RECREATE_TRIGGERS(ATABLE VARCHAR(128))
```

Description

RECREATE_TRIGGER is a utility procedure which recreates all the inoperative triggers defined against the table specified by **ASCHEMA** and **ATABLE**, using the SQL found in the system catalogue tables. It is useful for quickly recreating triggers which have been marked inoperative after a change to one or more dependencies. If **ASCHEMA** is omitted it defaults to the current schema.

Parameters

ASCHEMA If provided, the schema containing the table to recreate inoperative triggers for. If omitted, this parameter defaults to the value of the *CURRENT SCHEMA* special register.

ATRIGGER The name of the table to recreate inoperative triggers for.

Examples

Recreate all inoperative triggers defined against the *FINANCE.LEDGER* table:

```
CALL RECREATE_TRIGGERS('FINANCE', 'LEDGER');
```

Recreate all inoperative triggers defined against the *EMPLOYEE* table in the current schema:

```
CALL RECREATE_TRIGGERS('EMPLOYEE');
```

See Also

- Source code
- *RECREATE_TRIGGER procedure*
- SYSCAT.TRIGGERS (built-in catalogue view)

RECREATE_VIEW procedure

Recreates the specified inoperative view from its definition in the system catalogue.

Prototypes

```
RECREATE_VIEW(ASCHEMA VARCHAR(128), AVIEW VARCHAR(128))
RECREATE_VIEW(AVIEW VARCHAR(128))
```

Description

RECREATE_VIEW is a utility procedure which recreates the specified view using the SQL found in the system catalog tables. It is useful for quickly recreating views which have been marked inoperative after a change to one or more of the view's dependencies. If **ASCHEMA** is omitted it defaults to the current schema.

Note: This procedure is effectively redundant as of DB2 9.7 due to the new deferred revalidation functionality introduced in that version.

Warning: This procedure does *not* drop the view before recreating it. This guards against attempting to recreate an operative view (an inoperative view can be recreated without dropping it first). That said, it will not return an error in the case of attempting to recreate an operative view; the procedure will simply do nothing.

Warning: See *SAVE_AUTH procedure* for warnings regarding the loss of authorization information with inoperative views.

Parameters

ASCHEMA If provided, specifies the schema containing the view to recreate. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

AVIEW The name of the view to recreate.

Examples

Recreate the inoperative *FOO.BAR* view:

```
CALL RECREATE_VIEW('FOO', 'BAR');
```

Recreate the *BAZ* view in the current schema:

```
CALL RECREATE_VIEW('BAZ');
```

See Also

- Source code
- *RECREATE_VIEWS procedure*
- *SAVE_AUTH procedure*
- *SAVE_VIEW procedure*
- *RESTORE_VIEW procedure*
- SYSCAT.VIEWS (built-in catalog view)

RECREATE_VIEWS procedure

Recreates all inoperative views in the specified schema from their system catalogue definitions.

Prototypes

```
RECREATE_VIEWS(ASCHEMA VARCHAR(128))
RECREATE_VIEWS()
```

Description

RECREATE_VIEWS is a utility procedure which recreates all inoperative views in a specified schema, using the SQL found in the system catalogue tables. It is useful for quickly recreating views which have been marked inoperative after a change to one or more of the views' dependencies. If **ASCHEMA** is omitted it defaults to the current schema.

Note: This procedure is effectively redundant as of DB2 9.7 due to the new deferred revalidation functionality introduced in that version.

Warning: This procedure does not take into account the dependencies of views when recreating them. It crudely attempts to correctly order recreations on the basis of the CREATE_TIME field in the system catalogue, but this is not necessarily accurate. However, multiple consecutive runs of the procedure can be sufficient to recreate all inoperative views.

Warning: See *SAVE_AUTH procedure* for warnings regarding the loss of authorization information with inoperative views.

Parameters

ASCHEMA If provided, specifies the schema containing the views to recreate. If omitted, defaults to the value of the *CURRENT SCHEMA* special register.

Examples

Recreate all inoperative views in the *FOO* schema:

```
CALL RECREATE_VIEWS('FOO');
```

Recreate all inoperative views in the current schema:

```
CALL RECREATE_VIEWS;
```

See Also

- Source code
- *RECREATE_VIEW procedure*
- *SAVE_AUTH procedure*
- *SAVE_VIEW procedure*
- *RESTORE_VIEW procedure*
- SYSCAT.VIEWS (built-in catalog view)

REMOVE_AUTH procedure

Removes all authorities held by the specified name.

Prototypes

```
REMOVE_AUTH(AUTH_NAME VARCHAR(128), AUTH_TYPE VARCHAR(1), INCLUDE_PERSONAL VARCHAR(1))
REMOVE_AUTH(AUTH_NAME VARCHAR(128), INCLUDE_PERSONAL VARCHAR(1))
REMOVE_AUTH(AUTH_NAME VARCHAR(128))
```

Description

REMOVE_AUTH is a procedure which removes all authorizations from the entity specified by **AUTH_NAME**, and optionally **AUTH_TYPE**. If **AUTH_TYPE** is omitted *AUTH_TYPE scalar function* will be used to determine it. Otherwise, it must be `'U'`, `'G'`, or `'R'`, standing for user, group or role respectively.

Warning: This routine will not handle revoking column level authorizations, i.e. REFERENCES and UPDATES, which cannot be revoked directly but rather have to be revoked overall at the table level. Any such authorziations must be handled manually.

Parameters

AUTH_NAME The name of the user, group, or role to remove all authorizations from.

AUTH_TYPE The letter `'U'`, `'G'`, or `'R'` indicating whether **AUTH_NAME** refers to a user, group, or role respectively. If omitted, *AUTH_TYPE scalar function* will be used to determine the type of **AUTH_NAME**.

INCLUDE_PERSONAL If this parameter is `'Y'` and **AUTH_NAME** refers to a user, then all authorizations associated with the user's personal schema will be included. Defaults to `'N'` if omitted, meaning the user will still have access to all objects within their personal schema by default.

Examples

Remove all authorizations from the user *FRED*, but leave personal schema authorizations intact.

```
CALL REMOVE_AUTH('FRED');
```

Remove all authorizations from the *FINANCE* group (the **INCLUDE_PERSONAL** parameter is redundant here as **AUTH_NAME** is not a user).

```
CALL REMOVE_AUTH('FINANCE', 'G', 'N');
```

See Also

- Source code
- *AUTH_TYPE scalar function*
- *AUTHS_HELD table function*
- *AUTH_DIFF table function*
- *COPY_AUTH procedure*
- *MOVE_AUTH procedure*

RESTORE_AUTH procedure

Restores authorizations previously saved by *SAVE_AUTH procedure* for the specified table.

Prototypes

```
RESTORE_AUTH(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128))
RESTORE_AUTH(ATABLE VARCHAR(128))
```

Description

RESTORE_AUTH is a utility procedure which restores the authorization privileges for a table or view, previously saved by the *SAVE_AUTH procedure* procedure.

> **Warning:** Privileges may not be precisely restored. Specifically, the grantor in the restored privileges may be different to the original grantor if you are not the user that originally granted the privileges, or the original privileges were granted by the system. Furthermore, column specific authorizations (stored in SYSCAT.COLAUTH) are **not** saved and restored by these procedures.

Parameters

ASCHEMA The name of the schema containing the table for which authorizations are to be saved. If this parameter is omitted, it defaults to the value of the CURRENT SCHEMA special register.

ATABLE The name of the table within ASCHEMA for which authorizations are to be saved.

Examples

Save the authorizations associated with the FINANCE.LEDGER table, drop the table, recreate it with a definition derived from another table, then restore the authorizations:

```sql
SET SCHEMA FINANCE;
CALL SAVE_AUTH('LEDGER');
DROP TABLE LEDGER;
CREATE TABLE LEDGER LIKE LEDGER_TEMPLATE;
CALL RESTORE_AUTH('LEDGER');
```

Advanced usage: Copy the authorizations associated with FINANCE.SALES to FINANCE.SALES_HISTORY by changing the content of the SAVED_AUTH table (which is structured identically to the SYSCAT.TABAUTH table) between calls to *SAVE_AUTH procedure* and *RESTORE_AUTH procedure*:

```sql
SET SCHEMA FINANCE;
CALL SAVE_AUTH('SALES');
UPDATE UTILS.SAVED_AUTH
    SET TABNAME = 'SALES_HISTORY'
    WHERE TABNAME = 'SALES'
    AND TABSCHEMA = CURRENT SCHEMA;
CALL RESTORE_AUTH('SALES_HISTORY');
```

See Also

- Source code
- *SAVE_AUTH procedure*
- *SAVE_AUTHS procedure*
- *RESTORE_AUTHS procedure*
- SYSCAT.TABAUTH (built-in catalogue view)

RESTORE_AUTHS procedure

Restores the authorizations of all relations in the specified schema that were previously saved with *SAVE_AUTHS procedure*

Prototypes

```
RESTORE_AUTHS(ASCHEMA VARCHAR(128))
RESTORE_AUTHS()
```

Description

RESTORE_AUTHS is a utility procedure which restores the authorization settings (previously saved with *SAVE_AUTHS procedure*) for all tables in the specified schema. If no schema is specified, the current schema is used.

Warning: The procedure only attempts to restore settings for those tables or views which currently exist, and for which settings were previously saved. If you use *SAVE_AUTHS procedure* on a schema, drop several objects from the schema and then call *RESTORE_AUTHS procedure* on that schema, the procedure will succeed with no error, although several authorization settings have not been restored. Furthermore, any settings associated with the specified schema that are not restored are removed from store used by *SAVE_AUTHS procedure* (SAVED_AUTH in the schema containing the procedures).

Parameters

ASCHEMA The name of the schema containing the tables for which to restore authorziation settings. If this parameter is omitted the value of the *CURRENT SCHEMA* special register will be used instead.

Examples

Save all the authorization information from the tables in the *FINANCE_DEV* schema, do something arbitrary to the schema and restore the authorizations again:

```
SET SCHEMA FINANCE_DEV;
CALL SAVE_AUTHS();
-- Do something arbitrary to the schema (e.g. run a script to manipulate its␣
↪structure)
CALL RESTORE_AUTHS();
```

Advanced usage: Copy the authorizations from the *FINANCE_DEV* schema to the *FINANCE* schema by changing the content of SAVED_AUTH (this is the table in which *SAVE_AUTH procedure* temporarily stores authorizations; it has exactly the same structure as *SYSCAT.TABAUTH*):

```
CALL SAVE_AUTHS('FINANCE_DEV');
UPDATE UTILS.SAVED_AUTH
    SET TABSCHEMA = 'FINANCE'
    WHERE TABSCHEMA = 'FINANCE_DEV';
CALL RESTORE_AUTHS('FINANCE');
```

See Also

- Source code
- *SAVE_AUTH procedure*
- *SAVE_AUTHS procedure*
- *RESTORE_AUTH procedure*
- SYSCAT.TABAUTH (built-in catalogue view)

RESTORE_VIEW procedure

Restores the specified view which was previously saved with *SAVE_VIEW procedure*.

Prototypes

```
RESTORE_VIEW(ASCHEMA VARCHAR(128), AVIEW VARCHAR(128))
RESTORE_VIEW(AVIEW VARCHAR(128))
```

Description

RESTORE_VIEW is a utility procedure which restores the specified view using the SQL found in SAVED_VIEWS, which is populated initially by a call to *SAVE_VIEW procedure* or *SAVE_VIEWS procedure*. It also implicitly calls *RESTORE_AUTH procedure* to ensure that authorizations are not lost. This is the primary difference between using *SAVE_VIEW procedure* and RESTORE_VIEW, and using DB2's inoperative view mechanism with the *RECREATE_VIEW procedure* procedure.

Another use of these procedures is in recreating views which need to be dropped surrounding the update of a UDF.

Note: This procedure is effectively redundant as of DB2 9.7 due to the new deferred revalidation functionality introduced in that version.

Parameters

ASCHEMA If provided, the schema containing the view to restore. If omitted, this parameter defaults to the value of the *CURRENT SCHEMA* special register.

AVIEW The name of the view to restore.

Examples

Restore the definition of the *FINANCE.LEDGER_CHANGES* view:

```
CALL RESTORE_VIEW('FINANCE', 'LEDGER_CHANGES');
```

Restore the definition of the *EMPLOYEE_CHANGES* view in the current schema:

```
CALL RESTORE_VIEW('EMPLOYEE_CHANGES');
```

See Also

- Source code
- *SAVE_VIEW procedure*
- *RESTORE_VIEWS procedure*
- *RESTORE_AUTH procedure*
- SYSCAT.VIEWS (built-in catalogue view)

RESTORE_VIEWS procedure

Restores all views in the specified schema which were previously saved with *SAVE_VIEWS procedure*.

Prototypes

```
RESTORE_VIEWS(ASCHEMA VARCHAR(128))
RESTORE_VIEWS()
```

Description

RESTORE_VIEWS is a utility procedure which restores the definition of all views in the specified schema from SAVED_VIEWS which were previously stored with *SAVE_VIEW procedure* or *SAVE_VIEWS procedure*. RESTORE_VIEWS also implicitly calls *RESTORE_AUTH procedure* to restore the authorization of the views. This is in contrast to inoperative views recreated with *RECREATE_VIEWS procedure* which lose authorization information.

Note: This procedure is effectively redundant as of DB2 9.7 due to the new deferred revalidation functionality introduced in that version.

Parameters

ASCHEMA If provided, the schema containing the views to save. If omitted, this parameter defaults to the value of the *CURRENT SCHEMA* special register.

Examples

Restore the definition of all views in the *FINANCE* schema:

```
CALL RESTORE_VIEWS('FINANCE');
```

Restore the definition of all views in the current schema:

```
CALL RESTORE_VIEWS;
```

See Also

- Source code
- *SAVE_VIEWS procedure*
- *RESTORE_VIEW procedure*
- *RESTORE_AUTH procedure*
- SYSCAT.VIEWS (built-in catalogue view)

SAVE_AUTH procedure

Saves the authorizations of the specified relation for later restoration with *RESTORE_AUTH procedure*.

Prototypes

```
SAVE_AUTH(ASCHEMA VARCHAR(128), ATABLE VARCHAR(128))
SAVE_AUTH(ATABLE VARCHAR(128))
```

Description

SAVE_AUTH is a utility procedure which copies the authorization settings for the specified table or view from SYSCAT.TABAUTH to SAVED_AUTH (a utility table which exists in the same schema as the procedure). These saved settings can then be restored with the *RESTORE_AUTH procedure* procedure. These procedures are primarily intended for use in conjunction with the other schema evolution functions (like *RECREATE_VIEWS procedure*).

Warning: Column specific authorizations (stored in SYSCAT.COLAUTH) are *not* saved and restored by these procedures.

Note: *SAVE_AUTH procedure* and *RESTORE_AUTH procedure* are not used directly by *RECREATE_VIEW procedure* because when a view is marked inoperative, all authorization information is immediately wiped from *SYSCAT.TABAUTH*. Hence, there is nothing to restore by the time *RECREATE_VIEW procedure* is run.

You must call *SAVE_AUTH procedure* *before* performing the operation that will invalidate the view, and *RESTORE_AUTH procedure* *after* running *RECREATE_VIEW procedure*. Alternatively, you may wish to use *SAVE_VIEW procedure* and *RESTORE_VIEW procedure* instead, which rely on *SAVE_AUTH procedure* and *RESTORE_AUTH procedure* implicitly.

Parameters

ASCHEMA The name of the schema containing the table for which authorizations are to be saved. If this parameter is omitted, it defaults to the value of the *CURRENT SCHEMA* special register.

ATABLE The name of the table within **ASCHEMA** for which authorizations are to be saved.

Examples

Save the authorizations associated with the *FINANCE.LEDGER* table, drop the table, recreate it with a definition derived from another table, then restore the authorizations:

```
SET SCHEMA FINANCE;
CALL SAVE_AUTH('LEDGER');
DROP TABLE LEDGER;
CREATE TABLE LEDGER LIKE LEDGER_TEMPLATE;
CALL RESTORE_AUTH('LEDGER');
```

Advanced usage: Copy the authorizations associated with *FINANCE.SALES* to *FINANCE.SALES_HISTORY* by changing the content of the SAVED_AUTH table (which is structured identically to the *SYSCAT.TABAUTH* table) between calls to *SAVE_AUTH procedure* and *RESTORE_AUTH procedure*:

```
SET SCHEMA FINANCE;
CALL SAVE_AUTH('SALES');
UPDATE UTILS.SAVED_AUTH
    SET TABNAME = 'SALES_HISTORY'
    WHERE TABNAME = 'SALES'
    AND TABSCHEMA = CURRENT SCHEMA;
CALL RESTORE_AUTH('SALES_HISTORY');
```

See Also

- Source code
- *SAVE_AUTHS procedure*
- *SAVE_VIEW procedure*
- *RESTORE_AUTH procedure*
- *RESTORE_AUTHS procedure*
- *RESTORE_VIEW procedure*
- SYSCAT.TABAUTH (built-in catalogue view)

SAVE_AUTHS procedure

Saves the authorizations of all relations in the specified schema for later restoration with the *RESTORE_AUTHS procedure* procedure.

Prototypes

```
SAVE_AUTHS(ASCHEMA VARCHAR(128))
SAVE_AUTHS()
```

Description

SAVE_AUTHS is a utility procedure which copies the authorization settings for all tables in the specified schema. If no schema is specified the current schema is used. Essentially this is equivalent to running *SAVE_AUTH procedure* for every table in a schema.

Parameters

ASCHEMA The name of the schema containing the tables for which to save authorziation settings. If this parameter is omitted the value of the *CURRENT SCHEMA* special register will be used instead.

Examples

Save all the authorization information from the tables in the *FINANCE_DEV* schema, do something arbitrary to the schema and restore the authorizations again:

```
SET SCHEMA FINANCE_DEV;
CALL SAVE_AUTHS();
-- Do something arbitrary to the schema (e.g. run a script to manipulate its␣
↪structure)
CALL RESTORE_AUTHS();
```

Advanced usage: Copy the authorizations from the *FINANCE_DEV* schema to the FINANCE schema by changing the content of SAVED_AUTH (this is the table in which *SAVE_AUTH procedure* temporarily stores authorizations; it has exactly the same structure as *SYSCAT.TABAUTH*):

```
CALL SAVE_AUTHS('FINANCE_DEV');
UPDATE UTILS.SAVED_AUTH
    SET TABSCHEMA = 'FINANCE'
    WHERE TABSCHEMA = 'FINANCE_DEV';
CALL RESTORE_AUTHS('FINANCE');
```

See Also

- Source code
- *SAVE_AUTH procedure*
- *RESTORE_AUTH procedure*
- *RESTORE_AUTHS procedure*
- SYSCAT.TABAUTH (built-in catalogue view)

SAVE_VIEW procedure

Saves the authorizations and definition of the specified view for later restoration with *RESTORE_VIEW procedure*.

Prototypes

```
SAVE_VIEW(ASCHEMA VARCHAR(128), AVIEW VARCHAR(128))
SAVE_VIEW(AVIEW VARCHAR(128))
```

Description

SAVE_VIEW is a utility procedure which saves the definition of the specified view to SAVED_VIEWS. This saved definition can then be restored with the *RESTORE_VIEW procedure* procedure. SAVE_VIEW and RESTORE_VIEW also implicitly call *SAVE_AUTH procedure* and *RESTORE_AUTH procedure* to preserve the authorizations of the view. This is in contrast to inoperative views recreated with *RECREATE_VIEW procedure* which lose authorization information.

Note: This procedure is effectively redundant as of DB2 9.7 due to the new deferred revalidation functionality introduced in that version.

Parameters

ASCHEMA If provided, the schema containing the view to save. If omitted, this parameter defaults to the value of the *CURRENT SCHEMA* special register.

AVIEW The name of the view to save.

Examples

Save the definition of the *FINANCE.LEDGER_CHANGES* view:

```
CALL SAVE_VIEW('FINANCE', 'LEDGER_CHANGES');
```

Save the definition of the *EMPLOYEE_CHANGES* view in the current schema:

```
CALL SAVE_VIEW('EMPLOYEE_CHANGES');
```

See Also

- Source code
- *RESTORE_VIEW procedure*
- *SAVE_VIEWS procedure*
- *SAVE_AUTH procedure*
- SYSCAT.VIEWS (built-in catalogue view)

SAVE_VIEWS procedure

Saves the authorizations and definitions of all views in the specified schema for later restoration with *RESTORE_VIEWS procedure*.

Prototypes

```
SAVE_VIEWS(ASCHEMA VARCHAR(128))
SAVE_VIEWS()
```

Description

SAVE_VIEWS is a utility procedure which saves the definition of all views in the specified schema to SAVED_VIEWS. These saved definitions can then be restored with the *RESTORE_VIEWS procedure* procedure. SAVE_VIEWS also implicitly calls *SAVE_AUTH procedure* to preserve the authorizations of the views. This is in contrast to inoperative views recreated with *RECREATE_VIEW procedure* which lose authorization information.

Note: This procedure is effectively redundant as of DB2 9.7 due to the new deferred revalidation functionality introduced in that version.

Parameters

ASCHEMA If provided, the schema containing the views to save. If omitted, this parameter defaults to the value of the *CURRENT SCHEMA* special register.

Examples

Save the definition of all views in the *FINANCE* schema:

```
CALL SAVE_VIEWS('FINANCE');
```

Save the definition of all views in the current schema:

```
CALL SAVE_VIEWS;
```

See Also

- Source code
- *RESTORE_VIEWS procedure*
- *SAVE_VIEW procedure*
- *SAVE_AUTH procedure*
- SYSCAT.VIEWS (built-in catalogue view)

1.8 Change Log

1.8.1 Release 0.2 (XXX)

The second release mostly consisted of bug fixes and tidying up the documentation, but a couple of new features were introduced:

- The suite as a whole defines a couple of roles for management of the routines defined in the suite, and each module defines per-module subordinate roles allowing fine-grain control of who has access to which procedures
- The new assert.sql module includes a variety of routines for writing tests for the suite (and indeed databases in general)

- The new merge.sql module includes routines for automatically constructing “upsert” style MERGE statements (along with corresponding deletion and insertion statements) (#2)

1.8.2 Release 0.1 (2013-08-16)

First packaged release (despite the source repository being public for years :)

1.9 License

db2utils is distributed under the terms of the MIT license (an OSI approved license):

1.9.1 MIT License

1.9.2 Other Licenses

The MIT license is pretty permissive (typically it’s viewed as a “commercial friendly” license), but if anyone wants db2utils released under an additional open-source license (dual licensed), please feel free to contact me.

CHAPTER 2

Indices and tables

- genindex
- search