CLSQL Users' Guide

Kevin M. Rosenberg
Marcus T. Pearce
Pierre R. Mai
onShore Development, Inc.

CLSQL Users' Guide

by Kevin M. Rosenberg, Marcus T. Pearce, Pierre R. Mai, and onShore Development, Inc.

Table of Contents

Preface

This guide provides reference to the features of *CLSQL*. The first chapter provides an introduction to *CLSQL* and installation instructions. The reference sections document all user accessible symbols with examples of usage. There is a glossary of commonly used terms with their definitions.

Chapter 1. Introduction

Purpose

CLSQL is a Common Lisp interface to *SQL* databases. A number of Common Lisp implementations and SQL databases are supported. The general structure of *CLSQL* is based on the CommonSQL package by LispWorks Ltd.

History

The *CLSQL* project was started by Kevin M. Rosenberg in 2001 to support SQL access on multiple Common Lisp implementations using the *UFFI* library. The initial code was based substantially on Pierre R. Mai's excellent *MaiSQL* package. In late 2003, the UncommonSQL library was orphaned by its author, onShore Development, Inc. In April 2004, Marcus Pearce ported the UncommonSQL library to *CLSQL*. The UncommonSQL library provides a CommonSQL-compatible API for *CLSQL*.

The main changes from *MaiSQL* and UncommonSQL are:

- Port from the CMUCL FFI to *UFFI* which provide compatibility with the major Common Lisp implementations.
- Optimized loading of integer and floating-point fields.
- Additional database backends: ODBC, AODBC, SQLite version 2 and SQLite version 3.
- A compatibility layer for CMUCL specific code.
- Much improved robustness for the MySQL back-end along with version 4 client library support.
- Improved library loading and installation documentation.
- Improved packages and symbol export.
- Pooled connections.
- Integrated transaction support for the classic *MaiSQL* iteration macros.

Prerequisites

ASDF

CLSQL uses ASDF to compile and load its components. ASDF is included in the *CCLAN* [http://cclan.sourceforge.net] collection.

UFFI

CLSQL uses *UFFI* [http://uffi.kpe.io/] as a *Foreign Function Interface* (*FFI*) to support multiple ANSI Common Lisp implementations.

MD5

CLSQL's postgresql-socket interface uses Pierre Mai's md5 [http://files.kpe.io/md5/] module.

Supported Common Lisp Implementation

The implementations that support *CLSQL* is governed by the supported implementations of *UFFI*. The following implementations are supported:

- AllegroCL v6.2 through 8.0 on Debian Linux x86 & x86_64 & PowerPC, FreeBSD 4.5, and Microsoft Windows XP.
- Lispworks v4.3 and v4.4 on Debian Linux and Microsoft Windows XP.
- CMUCL 18e on Debian Linux, FreeBSD 4.5, and Solaris 2.8. 19c on Debian Linux.
- SBCL 0.8.4 through 0.9.16 on Debian Linux.
- SCL 1.1.1 on Debian Linux.
- OpenMCL 0.14 PowerPC and 1.0pre AMD64 on Debian Linux .

Supported SQL Implementation

CLSQL supports the following databases:

- MySQL (tested v3.23.51, v4.0.18, 5.0.24).
- PostgreSQL (tested with v7.4 and 8.0 with both direct API and TCP socket connections.
- SQLite version 2.
- SQLite version 3.
- Direct ODBC interface.
- Oracle OCI.
- Allegro's DB interface (AODBC).

Installation

Ensure ASDF is loaded

Simply load the file `asdf.lisp`.

```
(load "asdf.lisp")
```

Build C helper libraries

CLSQL uses functions that require 64-bit integer parameters and return values. The *FFI* in most *CLSQL* implementations do not support 64-bit integers. Thus, C helper libraries are required to break these 64-bit integers into two compatible 32-bit integers. The helper libraries reside in the directories `uffi` and `db-mysql`.

Microsoft Windows

Files named `Makefile.msvc` are supplied for building the libraries under Microsoft Windows. Since Microsoft Windows does not come with that compiler, compiled DLL and LIB library files are supplied with *CLSQL*.

UNIX

Files named `Makefile` are supplied for building the libraries under UNIX. Loading the `.asd` files automatically invokes make when necessary. So, manual building of the helper libraries is not necessary on most UNIX systems. However, the location of the MySQL library files and include files may need to adjusted in `db-mysql/Makefile` on non-Debian systems.

Add *UFFI* path

Unzip or untar the *UFFI* distribution which creates a directory for the *UFFI* files. Add that directory to ASDF's `asdf:*central-registry*`. You can do that by pushing the pathname of the directory onto this variable. The following example code assumes the *UFFI* files reside in the `/usr/share/lisp/uffi/` directory.

```
(push #P"/usr/share/lisp/uffi/" asdf:*central-registry*)
```

Add MD5 path

If you plan to use the clsql-postgresql-socket interface, you must load the md5 module. Unzip or untar the cl-md5 distribution, which creates a directory for the cl-md5 files. Add that directory to ASDF's `asdf:*central-registry*`. You can do that by pushing the pathname of the directory onto this variable. The following example code assumes the cl-md5 files reside in the `/usr/share/lisp/cl-md5/` directory.

```
(push #P"/usr/share/lisp/cl-md5/" asdf:*central-registry*)
```

Add *CLSQL* path and load module

Unzip or untar the *CLSQL* distribution which creates a directory for the *CLSQL* files. Add that directory to ASDF's `asdf:*central-registry*`. You can do that by pushing the pathname of the directory onto this variable. The following example code assumes the *CLSQL* files reside in the `/usr/share/lisp/clsql/` directory. You need to load the clsql system.

```
(push #P"/usr/share/lisp/clsql/" asdf:*central-registry*)
(asdf:operate 'asdf:load-op 'clsql)                  ; main CLSQL package
```

Run test suite (optional)

The test suite can be executed using the ASDF test-op operator. If *CLSQL* has not been loaded with asdf:load-op, the asdf:test-op operator will automatically load *CLSQL*. A configuration file named `.clsql-test.config` must be created in your home directory. There are instructures on the format

of that file in the `tests/README`. After creating `.clsql-test.config`, you can run the test suite with ASDF:

```
(asdf:operate 'asdf:test-op 'clsql)
```

Chapter 2. CommonSQL Tutorial

Based on the UncommonSQL Tutorial

Introduction

The goal of this tutorial is to guide a new developer thru the process of creating a set of *CLSQL* classes providing a Object-Oriented interface to persistent data stored in an SQL database. We will assume that the reader is familiar with how SQL works, how relations (tables) should be structured, and has created at least one SQL application previously. We will also assume a minor level of experience with Common Lisp.

CLSQL provides two different interfaces to SQL databases, a Functional interface, and an Object-Oriented interface. The Functional interface consists of a special syntax for embedded SQL expressions in Lisp, and provides lisp functions for SQL operations like SELECT and UPDATE. The object-oriented interface provides a way for mapping Common Lisp Objects System (CLOS) objects into databases and includes functions for inserting new objects, querying objects, and removing objects. Most applications will use a combination of the two.

CLSQL is based on the CommonSQL package from LispWorks Ltd, so the documentation that LispWorks makes available online is useful for *CLSQL* as well. It is suggested that developers new to *CLSQL* read their documentation as well, as any differences between CommonSQL and *CLSQL* are minor. LispWorks makes the following documents available:

- *Lispworks User Guide - The CommonSQL Package* [http://www.lispworks.com/documentation/lw44/LWUG/html/lwuser-204.htm]
- *Lispworks Reference Manual - The SQL Package* [http://www.lispworks.com/documentation/lw44/LWRM/html/lwref-424.htm]
- *CommonSQL Tutorial by Nick Levine* [http://www.lispworks.com/documentation/sql-tutorial/index.html]

Data Modeling with *CLSQL*

Before we can create, query and manipulate *CLSQL* objects, we need to define our data model as noted by Philip Greenspun [1]

When data modeling, you are telling the relational database management system (RDBMS) the following:

- What elements of the data you will store.
- How large each element can be.
- What kind of information each element can contain.
- What elements may be left blank.
- Which elements are constrained to a fixed range.
- Whether and how various tables are to be linked.

[1] *Philip Greenspun's "SQL For Web Nerds" - Data Modeling* [http://philip.greenspun.com/sql/data-modeling.html]

With SQL database one would do this by defining a set of relations, or tables, followed by a set of queries for joining the tables together in order to construct complex records. However, with *CLSQL* we do this by defining a set of CLOS classes, specifying how they will be turned into tables, and how they can be joined to one another via relations between their attributes. The SQL tables, as well as the queries for joining them together are created for us automatically, saving us from dealing with some of the tedium of SQL.

Let us start with a simple example of two SQL tables, and the relations between them.

```
CREATE TABLE EMPLOYEE ( emplid     NOT NULL number(38),
                        first_name NOT NULL varchar2(30),
                        last_name  NOT NULL varchar2(30),
                        email              varchar2(100),
                        companyid  NOT NULL number(38),
                        managerid          number(38))

CREATE TABLE COMPANY ( companyid   NOT NULL number(38),
                       name        NOT NULL varchar2(100),
                       presidentid NOT NULL number(38))
```

This is of course the canonical SQL tutorial example, "The Org Chart".

In *CLSQL*, we would have two "view classes" (a fancy word for a class mapped into a database). They would be defined as follows:

```
(clsql:def-view-class employee ()
  ((emplid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :emplid)
   (first-name
    :accessor first-name
    :type (string 30)
    :initarg :first-name)
   (last-name
    :accessor last-name
    :type (string 30)
    :initarg :last-name)
   (email
    :accessor employee-email
    :type (string 100)
    :nulls-ok t
    :initarg :email)
   (companyid
    :type integer
    :initarg :companyid)
   (managerid
    :type integer
    :nulls-ok t
    :initarg :managerid))
  (:base-table employee))

(clsql:def-view-class company ()
```

```
((companyid
  :db-kind :key
  :db-constraints :not-null
  :type integer
  :initarg :companyid)
 (name
  :type (string 100)
  :initarg :name)
 (presidentid
  :type integer
  :initarg :presidentid))
(:base-table company))
```

The `DEF-VIEW-CLASS` macro is just like the normal CLOS `DEFCLASS` macro, except that it handles several slot options that `DEFCLASS` doesn't. These slot options have to do with the mapping of the slot into the database. We only use a few of the slot options in the above example, but there are several others.

- :column - The name of the SQL column this slot is stored in. Defaults to the slot name. If the slot name is not a valid SQL identifier, it is escaped, so foo-bar becomes foo_bar.
- :db-kind - The kind of database mapping which is performed for this slot. :base indicates the slot maps to an ordinary column of the database view. :key indicates that this slot corresponds to part of the unique keys for this view, :join indicates a join slot representing a relation to another view and :virtual indicates that this slot is an ordinary CLOS slot. Defaults to :base.
- :db-reader - If a string, then when reading values from the database, the string will be used for a format string, with the only value being the value from the database. The resulting string will be used as the slot value. If a function then it will take one argument, the value from the database, and return the value that should be put into the slot.
- :db-writer - If a string, then when reading values from the slot for the database, the string will be used for a format string, with the only value being the value of the slot. The resulting string will be used as the column value in the database. If a function then it will take one argument, the value of the slot, and return the value that should be put into the database.
- :db-type - A string which will be used as the type specifier for this slots column definition in the database.
- :void-value - The Lisp value to return if the field is `NULL`. The default is `NIL`.
- :db-info - A join specification.

In our example each table as a primary key attribute, which is required to be unique. We indicate that a slot is part of the primary key (*CLSQL* supports multi-field primary keys) by specifying the :db-kind key slot option.

The SQL type of a slot when it is mapped into the database is determined by the :type slot option. The argument for the :type option is a Common Lisp datatype. The *CLSQL* framework will determine the appropriate mapping depending on the database system the table is being created in. If we really wanted to determine what SQL type was used for a slot, we could specify a :db-type option like "NUMBER(38)" and we would be guaranteed that the slot would be stored in the database as a NUMBER(38). This is not recomended because it could makes your view class unportable across database systems.

`DEF-VIEW-CLASS` also supports some class options, like :base-table. The :base-table option specifies what the table name for the view class will be when it is mapped into the database.

Another class option is :normalizedp, which signals *CLSQL* to use a normalized schema for the mapping from slots to SQL columns. By default *CLSQL* includes all the slots of a parent class that map to SQL

columns into the child class. This option tells *CLSQL* to normalize the schema, so that a join is done on the primary keys of the concerned tables to get a complete column set for the classes. For more information, see `def-view-class`.

Class Relations

In an SQL only application, the EMPLOYEE and COMPANY tables can be queried to determine things like, "Who is Vladimir's manager?", "What company does Josef work for?", and "What employees work for Widgets Inc.". This is done by joining tables with an SQL query.

Who works for Widgets Inc.?

```
SELECT first_name, last_name FROM employee, company
       WHERE employee.companyid = company.companyid
      AND company.company_name = "Widgets Inc."
```

Who is Vladimir's manager?

```
SELECT managerid FROM employee
       WHERE employee.first_name = "Vladimir"
      AND employee.last_name = "Lenin"
```

What company does Josef work for?

```
SELECT company_name FROM company, employee
       WHERE employee.first_name = "Josef"
      AND employee.last-name = "Stalin"
      AND employee.companyid = company.companyid
```

With *CLSQL* however we do not need to write out such queries because our view classes can maintain the relations between employees and companies, and employees to their managers for us. We can then access these relations like we would any other attribute of an employee or company object. In order to do this we define some join slots for our view classes.

What company does an employee work for? If we add the following slot definition to the employee class we can then ask for it's COMPANY slot and get the appropriate result.

```
    ;; In the employee slot list
    (company
      :accessor employee-company
      :db-kind :join
      :db-info (:join-class company
         :home-key companyid
  :foreign-key companyid
  :set nil))
```

Who are the employees of a given company? And who is the president of it? We add the following slot definition to the company view class and we can then ask for it's EMPLOYEES slot and get the right result.

```
      ;; In the company slot list
      (employees
```

```
:reader company-employees
:db-kind :join
:db-info (:join-class employee
   :home-key companyid
   :foreign-key companyid
   :set t))

      (president
       :reader president
:db-kind :join
:db-info (:join-class employee
   :home-key presidentid
   :foreign-key emplid
   :set nil))
```

And lastly, to define the relation between an employee and their manager:

```
;; In the employee slot list
      (manager
       :accessor employee-manager
:db-kind :join
:db-info (:join-class employee
          :home-key managerid
   :foreign-key emplid
   :set nil))
```

CLSQL join slots can represent one-to-one, one-to-many, and many-to-many relations. Above we only have one-to-one and one-to-many relations, later we will explain how to model many-to-many relations. First, let's go over the slot definitions and the available options.

In order for a slot to be a join, we must specify that it's :db-kind :join, as opposed to :base or :key. Once we do that, we still need to tell *CLSQL* how to create the join statements for the relation. This is what the :db-info option does. It is a list of keywords and values. The available keywords are:

- :join-class - The view class to which we want to join. It can be another view class, or the same view class as our object.
- :home-key - The slot(s) in the immediate object whose value will be compared to the foreign-key slot(s) in the join-class in order to join the two tables. It can be a single slot-name, or it can be a list of slot names.
- :foreign-key - The slot(s) in the join-class which will be compared to the value(s) of the home-key.
- :set - A boolean which if false, indicates that this is a one-to-one relation, only one object will be returned. If true, than this is a one-to-many relation, a list of objects will be returned when we ask for this slots value.

There are other :join-info options available in *CLSQL*, but we will save those till we get to the many-to-many relation examples.

Object Oriented Class Relations

CLSQL provides an Object Oriented Data Definition Language, which provides a mapping from SQL tables to CLOS objects. By default class inheritance is handled by including all the columns from parent classes into the child class. This means your database schema becomes very much denormalized. The class

option :normalizedp can be used to disable the default behaviour and have *CLSQL* normalize the database schemas of inherited classes.

See `def-view-class` for more information.

Object Creation

Now that we have our model laid out, we should create some object. Let us assume that we have a database connect set up already. We first need to create our tables in the database:

Note: the file `examples/clsql-tutorial.lisp` contains view class definitions which you can load into your list at this point in order to play along at home.

```
(clsql:create-view-from-class 'employee)
(clsql:create-view-from-class 'company)
```

Then we will create our objects. We create them just like you would any other CLOS object:

```
(defvar company1 (make-instance 'company
         :companyid 1
         :presidentid 1
         :name "Widgets Inc."))

(defvar employee1 (make-instance 'employee
          :emplid 1
          :first-name "Vladimir"
          :last-name "Lenin"
          :email "lenin@soviet.org"
          :companyid 1))

(defvar employee2 (make-instance 'employee
          :emplid 2
          :first-name "Josef"
          :last-name "Stalin"
          :email "stalin@soviet.org"
          :companyid 1
          :managerid 1))
```

In order to insert an objects into the database we use the `UPDATE-RECORDS-FROM-INSTANCE` function as follows:

```
(clsql:update-records-from-instance employee1)
(clsql:update-records-from-instance employee2)
(clsql:update-records-from-instance company1)
```

After you make any changes to an object, you have to specifically tell *CLSQL* to update the SQL database. The `UPDATE-RECORDS-FROM-INSTANCE` method will write all of the changes you have made to the object into the database.

Since *CLSQL* objects are just normal CLOS objects, we can manipulate their slots just like any other object. For instance, let's say that Lenin changes his email because he was getting too much spam from the German Socialists.

```
;; Print Lenin's current email address, change it and save it to the
;; database.  Get a new object representing Lenin from the database
;; and print the email

;; This lets us use the functional CLSQL interface with [] syntax
(clsql:locally-enable-sql-reader-syntax)

(format t "The email address of ~A ~A is ~A"
 (first-name employee1)
 (last-name employee1)
 (employee-email employee1))

(setf (employee-email employee1) "lenin-nospam@soviets.org")

;; Update the database
(clsql:update-records-from-instance employee1)

(let ((new-lenin (car (clsql:select 'employee
                 :where [= [slot-value 'employee 'emplid] 1]))))
      (format t "His new email is ~A"
   (employee-email new-lenin)))
```

Everything except for the last `LET` expression is already familiar to us by now. To understand the call to `CLSQL:SELECT` we need to discuss the Functional SQL interface and it's integration with the Object Oriented interface of *CLSQL*.

Finding Objects

Now that we have our objects in the database, how do we get them out when we need to work with them? *CLSQL* provides a functional interface to SQL, which consists of a special Lisp reader macro and some functions. The special syntax allows us to embed SQL in lisp expressions, and lisp expressions in SQL, with ease.

Once we have turned on the syntax with the expression:

```
(clsql:locally-enable-sql-reader-syntax)
```

We can start entering fragments of SQL into our lisp reader. We will get back objects which represent the lisp expressions. These objects will later be compiled into SQL expressions that are optimized for the database backed we are connected to. This means that we have a database independent SQL syntax. Here are some examples:

```
;; an attribute or table name
[foo] => #<CLSQL-SYS::SQL-IDENT-ATTRIBUTE FOO>

;; a attribute identifier with table qualifier
[foo bar] => #<CLSQL-SYS::SQL-IDENT-ATTRIBUTE FOO.BAR>

;; a attribute identifier with table qualifier
[= "Lenin" [first_name]] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP ('Lenin' = FIRST_NAME)>
```

```
[< [emplid] 3] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP (EMPLID < 3)>

[and [< [emplid] 2] [= [first_name] "Lenin"]] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP ((EMPLID < 2) AND
                                    (FIRST_NAME = 'Lenin'))>


;; If we want to reference a slot in an object we can us the
;;  SLOT-VALUE sql extension
[= [slot-value 'employee 'emplid] 1] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP (EMPLOYEE.EMPLID = 1)>

[= [slot-value 'employee 'emplid]
   [slot-value 'company 'presidentid]] =>
   #<CLSQL-SYS::SQL-RELATIONAL-EXP (EMPLOYEE.EMPLID = COMPANY.PRESIDENTID)>
```

The `SLOT-VALUE` operator is important because it let's us query objects in a way that is robust to any changes in the object->table mapping, like column name changes, or table name changes. So when you are querying objects, be sure to use the `SLOT-VALUE` SQL extension.

Since we can now formulate SQL relational expression which can be used as qualifiers, like we put after the WHERE keyword in SQL statements, we can start querying our objects. *CLSQL* provides a function SELECT which can return use complete objects from the database which conform to a qualifier, can be sorted, and various other SQL operations.

The first argument to SELECT is a class name. it also has a set of keyword arguments which are covered in the documentation. For now we will concern ourselves only with the :where keyword. Select returns a list of objects, or nil if it can't find any. It's important to remember that it always returns a list, so even if you are expecting only one result, you should remember to extract it from the list you get from SELECT.

```
;; all employees
(clsql:select 'employee)
;; all companies
(clsql:select 'company)

;; employees named Lenin
(clsql:select 'employee :where [= [slot-value 'employee 'last-name]
    "Lenin"])

(clsql:select 'company :where [= [slot-value 'company 'name]
          "Widgets Inc."])

;; Employees of Widget's Inc.
(clsql:select 'employee
     :where [and [= [slot-value 'employee 'companyid]
      [slot-value 'company 'companyid]]
   [= [slot-value 'company 'name]
      "Widgets Inc."]])

;; Same thing, except that we are using the employee
;; relation in the company view class to do the join for us,
;; saving us the work of writing out the SQL!
```

```
(company-employees company1)

;; President of Widgets Inc.
(president company1)

;; Manager of Josef Stalin
(employee-manager employee2)
```

Deleting Objects

Now that we know how to create objects in our database, manipulate them and query them (including using our predefined relations to save us the trouble writing alot of SQL) we should learn how to clean up after ourself. It's quite simple really. The function `DELETE-INSTANCE-RECORDS` will remove an object from the database. However, when we remove an object we are responsible for making sure that the database is left in a correct state.

For example, if we remove a company record, we need to either remove all of it's employees or we need to move them to another company. Likewise if we remove an employee, we should make sure to update any other employees who had them as a manager.

Conclusion

There are many nooks and crannies to *CLSQL*, some of which are covered in the Xanalys documents we refered to earlier, some are not. The best documentation at this time is still the source code for *CLSQL* itself and the inline documentation for its various functions.

Connection and Initialisation

This section describes the *CLSQL* interface for initialising database interfaces of different types, creating and destroying databases and connecting and disconnecting from databases.

Table of Contents

Name

DATABASE — The super-type of all *CLSQL* databases
Class

Class Precedence List

database, standard-object, t

Description

This class is the superclass of all *CLSQL* databases. The different database back-ends derive subclasses of this class to implement their databases. No instances of this class are ever created by *CLSQL*.

Name

CONNECT-IF-EXISTS — Default value for the *if-exists* parameter of connect.
Variable

Value Type

A valid argument to the *if-exists* parameter of connect, that is, one of :new, :warn-new, :error, :warn-old, :old.

Initial Value

:error

Description

The value of this variable is used in calls to connect as the default value of the *if-exists* parameter. See connect for the semantics of the valid values for this variable.

Examples

None.

Affected By

None.

See Also

connect

Notes

None.

Name

DB-POOL-MAX-FREE-CONNECTIONS — How many free connections should the connection pool try to keep.
Parameter

Value Type

Integer

Initial Value

4

Description

Threshold of free-connections in the pool before we disconnect a database rather than returning it to the pool. NIL for no limit. This is really a heuristic that should, on avg keep the free connections about this size.

Note

This is not a hard limit, the number of connections in the pool may exceed this value.

Examples

```
(setf clsql-sys:*db-pool-max-free-connections* 2)
```

Affected By

None

See Also

`connect`
`disconnect`

Notes

Note

Name

DEFAULT-DATABASE — The default database object to use.
Variable

Value Type

Any object of type database, or `NIL` to indicate no default database.

Initial Value

`NIL`

Description

Any function or macro in *CLSQL* that operates on a database uses the value of this variable as the default value for it's *`database`* parameter.

The value of this parameter is changed by calls to `connect`, which sets *default-database* to the database object it returns. It is also changed by calls to `disconnect`, when the database object being disconnected is the same as the value of *default-database*. In this case `disconnect` sets *default-database* to the first database that remains in the list of active databases as returned by `connected-databases`, or `NIL` if no further active databases exist.

The user may change *default-database* at any time to a valid value of his choice.

Caution

If the value of *default-database* is `NIL`, then all calls to *CLSQL* functions on databases must provide a suitable *`database`* parameter, or an error will be signalled.

Examples

```
(connected-databases)
=> NIL
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48385F55}>
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {483868FD}>
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql :if-exist
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48387265}>
*default-database*
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48387265}>
(disconnect)
=> T
*default-database*
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {483868FD}>
(disconnect)
=> T
*default-database*
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48385F55}>
(disconnect)
```

```
=> T
*default-database*
=> NIL
(connected-databases)
=> NIL
```

Affected By

`connect`
`disconnect`

See Also

`connected-databases`

Notes

Note

This variable is intended to facilitate working with *CLSQL* in an interactive fashion at the top-level loop, and because of this, `connect` and `disconnect` provide some fairly complex behaviour to keep *default-database* set to useful values. Programmatic use of *CLSQL* should never depend on the value of *default-database* and should provide correct database objects via the *`database`* parameter to functions called.

Name

DEFAULT-DATABASE-TYPE — The default database type to use
Variable

Value Type

Any keyword representing a valid database back-end of *CLSQL*, or `NIL`.

Initial Value

`NIL`

Description

The value of this variable is used in calls to `initialize-database-type` and `connect` as the default value of the *`database-type`* parameter.

Caution

If the value of this variable is `NIL`, then all calls to `initialize-database-type` or `connect` will have to specify the *`database-type`* to use, or a general-purpose error will be signalled.

Examples

```
(setf *default-database-type* :mysql)
=> :mysql
(initialize-database-type)
=> t
```

Affected By

None.

See Also

`intitialize-database-type`

Notes

None.

Name

INITIALIZED-DATABASE-TYPES — List of all initialized database types
Variable

Value Type

A list of all initialized database types, each of which represented by it's corresponding keyword.

Initial Value

`NIL`

Description

This variable is updated whenever `initialize-database-type` is called for a database type which hasn't already been initialized before, as determined by this variable. In that case the keyword representing the database type is pushed onto the list stored in *INITIALIZED-DATABASE-TYPES*.

Caution

Attempts to modify the value of this variable will result in undefined behaviour.

Examples

```
(setf *default-database-type* :mysql)
=> :mysql
(initialize-database-type)
=> t
*initialized-database-types*
=> (:MYSQL)
```

Affected By

`initialize-database-type`

See Also

`intitialize-database-type`

Notes

Direct access to this variable is primarily provided because of compatibility with Harlequin's Common SQL.

Name

CONNECT — create a connection to a database.
Function

Syntax

```
connect connection-spec &key if-exists database-type pool make-default => database
```

Arguments and Values

connection-spec	A SQL backend specific connection specification supplied as a list or as a string. For the MySQL backend, this list includes an optional associative list of connection options. The options list is parsed and supplied to the MySQL API using `mysql_options` in between the calls to `mysql_init` and `mysql_real_connect`.
if-exists	This indicates the action to take if a connection to the same database exists already. See below for the legal values and actions. It defaults to the value of *connect-if-exists*.
database-type	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
pool	A boolean flag. If `T`, acquire connection from a pool of open connections. If the pool is empty, a new connection is created. The default is `NIL`.
make-default	A boolean flag. If `T`, *default-database* is set to the new connection, otherwise *default-database* is not changed. The default is `T`.
database	The database object representing the connection.

Description

This function takes a connection specification and a database type and creates a connection to the database specified by those. The type and structure of the connection specification depend on the database type.

The parameter *if-exists* specifies what to do if a connection to the database specified exists already, which is checked by calling `find-database` on the database name returned by `database-name-from-spec` when called with the *connection-spec* and *database-type* parameters. The possible values of *if-exists* are:

:new	Go ahead and create a new connection.
:warn-new	This is just like :new, but also signals a warning of type clsql-exists-warning, indicating the old and newly created databases.
:error	This will cause `connect` to signal a correctable error of type clsql-exists-error. The user may choose to proceed, either by indicating that a new connection shall be created, via the restart create-new, or by indicating that the existing connection shall be used, via the restart use-old.
:old	This will cause `connect` to use an old connection if one exists.
:warn-old	This is just like :old, but also signals a warning of type clsql-exists-warning, indicating the old database used, via the slots old-db and new-db

The database name of the returned database object will be the same under `string=` as that which would be returned by a call to `database-name-from-spec` with the given *`connection-spec`* and *`database-type`* parameters.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48036F6D}>
(database-name *)
=> "dent/newesim/dent"

(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
>> In call to CONNECT:
>>   There is an existing connection #<CLSQL-MYSQL:MYSQL-DATABASE {48036F6D}> to d
>>
>> Restarts:
>>   0: [CREATE-NEW] Create a new connection.
>>   1: [USE-OLD   ] Use the existing connection.
>>   2: [ABORT     ] Return to Top-Level.
>>
>> Debug  (type H for help)
>>
>> (CONNECT ("dent" "newesim" "dent" "dent") :IF-EXISTS NIL :DATABASE-TYPE ...)
>> Source:
>> ; File: /prj/CLSQL/sql/sql.cl
>> (RESTART-CASE (ERROR 'CLSQL-EXISTS-ERROR :OLD-DB OLD-DB)
>>               (CREATE-NEW NIL :REPORT "Create a new connection."
>>                (SETQ RESULT #))
>>               (USE-OLD NIL :REPORT "Use the existing connection."
>>                (SETQ RESULT OLD-DB)))
>> 0] 0
=> #<CLSQL-MYSQL:MYSQL-DATABASE {480451F5}>
```

Side Effects

A database connection is established, and the resultant database object is registered, so as to appear in the list returned by `connected-databases`. *default-database* may be rebound to the created object.

Affected by

default-database-type
connect-if-exists

Exceptional Situations

If the connection specification is not syntactically or semantically correct for the given database type, an error of type sql-user-error is signalled. If during the connection attempt an error is detected (e.g. because of permission problems, network trouble or any other cause), an error of type sql-database-error is signalled.

If a connection to the database specified by *connection-spec* exists already, conditions are signalled according to the *if-exists* parameter, as described above.

See Also

connected-databases
disconnect
reconnect
connect-if-exists
find-database
status

Notes

The *pool* and *make-default* keyword arguments to connect are *CLSQL* extensions.

Name

CONNECTED-DATABASES — Return the list of active database objects.
Function

Syntax

```
connected-databases => databases
```

Arguments and Values

databases The list of active database objects.

Description

This function returns the list of active database objects, i.e. all those database objects created by calls to `connect`, which have not been closed by calling `disconnect` on them.

Caution

The consequences of modifying the list returned by `connected-databases` are undefined.

Examples

```
(connected-databases)
=> NIL
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {4830BC65}>
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {4830C5AD}>
(connected-databases)
=> (#<CLSQL-MYSQL:MYSQL-DATABASE {4830C5AD}>
    #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {4830BC65}>)
(disconnect)
=> T
(connected-databases)
=> (#<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {4830BC65}>)
(disconnect)
=> T
(connected-databases)
=> NIL
```

Side Effects

None.

Affected By

`connect`

```
disconnect
```

Exceptional Situations

None.

See Also

```
disconnect
connect
status
find-database
```

Notes

None.

Name

DATABASE-NAME — Get the name of a database object
Generic Function

Syntax

`database-name` *`database`* `=> name`

Arguments and Values

`database` A database object, either of type database or of type closed-database.

name A string describing the identity of the database to which this database object is connected to.

Description

This function returns the database name of the given database. The database name is a string which somehow describes the identity of the database to which this database object is or has been connected. The database name of a database object is determined at `connect` time, when a call to `database-name-from-spec` derives the database name from the connection specification passed to `connect` in the *`connection-spec`* parameter.

The database name is used via `find-database` in `connect` to determine whether database connections to the specified database exist already.

Usually the database name string will include indications of the host, database name, user, or port that where used during the connection attempt. The only important thing is that this string shall try to identify the database at the other end of the connection. Connection specifications parts like passwords and credentials shall not be used as part of the database name.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48391DCD}>
(database-name *default-database*)
=> "dent/newesim/dent"

(database-name-from-spec '(nil "template1" "dent" nil) :postgresql)
=> "/template1/dent"
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(database-name *default-database*)
=> "/template1/dent"

(database-name-from-spec '("www.pmsf.de" "template1" "dent" nil) :postgresql)
=> "www.pmsf.de/template1/dent"
```

Side Effects

None.

Affected By

`database-name-from-spec`

Exceptional Situations

Will signal an error if the object passed as the *`database`* parameter is neither of type database nor of type closed-database.

See Also

`connect`
`find-database`
`connected-databases`
`disconnect`
`status`

Notes

None.

Name

DATABASE-NAME-FROM-SPEC — Return the database name string corresponding to the given connection specification.
Generic Function

Syntax

```
database-name-from-spec connection-spec database-type => name
```

Arguments and Values

connection-spec	A connection specification, whose structure and interpretation are dependent on the *database-type*.
database-type	A database type specifier, i.e. a keyword.
name	A string denoting a database name.

Description

This generic function takes a connection specification and a database type and returns the database name of the database object that would be created had `connect` been called with the given connection specification and database types.

This function is useful in determining a database name from the connection specification, since the way the connection specification is converted into a database name is dependent on the database type.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48391DCD}>
(database-name *default-database*)
=> "dent/newesim/dent"

(database-name-from-spec '(nil "template1" "dent" nil) :postgresql)
=> "/template1/dent"
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(database-name *default-database*)
=> "/template1/dent"

(database-name-from-spec '("www.pmsf.de" "template1" "dent" nil) :postgresql)
=> "www.pmsf.de/template1/dent"

(find-database "dent/newesim/dent")
=> #<CLSQL-MYSQL:MYSQL-DATABASE {484E91C5}>
(find-database "/template1/dent")
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
```

```
(find-database "www.pmsf.de/template1/dent" nil)
=> NIL
(find-database **)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
```

Side Effects

None.

Affected by

None.

Exceptional Situations

If the value of *connection-spec* is not a valid connection specification for the given database type, an error of type clsql-invalid-spec-error might be signalled.

See Also

connect

Notes

database-name-from-spec is a *CLSQL* extension.

Name

DATABASE-TYPE — Get the type of a database object.
Generic Function

Syntax

```
database-type DATABASE => type
```

Arguments and Values

database	A database object, either of type database or of type closed-database.
type	A keyword symbol denoting a known database back-end.

Description

Returns the type of *database*.

Examples

```
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(database-type *default-database*)
=> :postgresql
```

Side Effects

None.

Affected by

None.

Exceptional Situations

Will signal an error if the object passed as the *database* parameter is neither of type database nor of type closed-database.

See Also

```
connect
find-database
connected-databases
disconnect
status
```

Notes

`database-type` is a *CLSQL* extension.

Name

DISCONNECT — close a database connection
Function

Syntax

```
disconnect &key database error => result
```

Arguments and Values

error	A boolean flag indicating whether to signal an error if *database* is non-NIL but cannot be found.
database	The database to disconnect, which defaults to the database indicated by *default-database*.
result	A Boolean indicating whether a connection was successfully disconnected.

Description

This function takes a database object as returned by connect, and closes the connection. If no matching database is found and *error* and *database* are both non-NIL an error is signaled, otherwise NIL is returned. If the database is from a pool it will be released to this pool.

The status of the object passed is changed to closed after the disconnection succeeds, thereby preventing further use of the object as an argument to *CLSQL* functions, with the exception of database-name and database-type. If the user does pass a closed database to any other *CLSQL* function, an error of type sql-fatal-error is signalled.

Examples

```
(disconnect :database (find-database "dent/newesim/dent"))
=> T
```

Side Effects

The database object is removed from the list of connected databases as returned by connected-databases.

If the database object passed is the same under eq as the value of *default-database*, then *default-database* is set to the first remaining database from connected-databases or to NIL if no further active database exists.

Non-pooled

The database connection is closed and the state of the database object is changed to closed.

Pooled

Unless there are already *db-pool-max-free-connections* free connections in the pool it is returned to the pool, with the backend having an opportunity to run generic cleanup on the connection first. If the max free connections has already been reached then it is disconnected as if it were not in the pool.

Affected by

default-database
db-pool-max-free-connections

Exceptional Situations

If during the disconnection attempt an error is detected (e.g. because of network trouble or any other cause), an error of type sql-error might be signalled.

See Also

`connect`
`disconnect-pooled`

Notes

None.

Name

DISCONNECT-POOLED — closes all pooled database connections
Function

Syntax

```
disconnect-pooled &optional clear => t
```

Description

This function disconnects all database connections that have been placed into the pool by calling `connect` with :pool `T`.

If optional argument clear is non-`NIL` then the connection-pool objects are also removed.

Examples

```
(disconnect-pool)
=> T
```

Side Effects

Database connections will be closed and *all* entries in the pool are removed. This is done with great prejudice and no thought to thread safety or whether that connection is currently in use.

Affected by

`disconnect`

Exceptional Situations

If during the disconnection attempt an error is detected (e.g. because of network trouble or any other cause), an error of type clsql-error might be signalled.

See Also

`connect`
`disconnect`

Notes

`disconnect-pooled` is a *CLSQL* extension.

Name

FIND-DATABASE — >Locate a database object through it's name.
Function

Syntax

```
find-database database &optional errorp => result
```

Arguments and Values

database	A database object or a string, denoting a database name.
errorp	A generalized boolean. Defaults to t.
db-type	A keyword symbol denoting a known database back-end.
result	Either a database object, or, if *errorp* is NIL, possibly NIL.

Description

find-database locates an active database object given the specification in *database*. If *database* is an object of type database, find-database returns this. Otherwise it will search the active databases as indicated by the list returned by connected-databases for a database of type *db-type* whose name (as returned by database-name is equal as per string= to the string passed as *database*. If it succeeds, it returns the first database found.

If *db-type* is NIL all databases matching the string *database* are considered. If no matching databases are found and *errorp* is NIL then NIL is returned. If *errorp* is NIL and one or more matching databases are found, then the most recently connected database is returned as a first value and the number of matching databases is returned as a second value. If no, or more than one, matching databases are found and *errorp* is true, an error is signalled.

Examples

```
(database-name-from-spec '("dent" "newesim" "dent" "dent") :mysql)
=> "dent/newesim/dent"
(connect '("dent" "newesim" "dent" "dent") :database-type :mysql)
=> #<CLSQL-MYSQL:MYSQL-DATABASE {48391DCD}>
(database-name *default-database*)
=> "dent/newesim/dent"

(database-name-from-spec '(nil "template1" "dent" nil) :postgresql)
=> "/template1/dent"
(connect '(nil "template1" "dent" nil) :database-type :postgresql)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(database-name *default-database*)
=> "/template1/dent"

(database-name-from-spec '("www.pmsf.de" "template1" "dent" nil) :postgresql)
=> "www.pmsf.de/template1/dent"
```

```
(find-database "dent/newesim/dent")
=> #<CLSQL-MYSQL:MYSQL-DATABASE {484E91C5}>
(find-database "/template1/dent")
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
(find-database "www.pmsf.de/template1/dent" nil)
=> NIL
(find-database **)
=> #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {48392D2D}>
```

Side Effects

None.

Affected By

`connected-databases`

Exceptional Situations

Will signal an error of type clsql-error if no matching database can be found, and *`errorp`* is true. Will signal an error if the value of *`database`* is neither an object of type database nor a string.

See Also

`database-name`
`database-name-from-spec`
`disconnect`
`connect`
`status`
`connected-databases`

Notes

The *`db-type`* keyword argument to `find-database` is a *CLSQL* extension.

Name

INITIALIZE-DATABASE-TYPE — Initializes a database type
Function

Syntax

```
initialize-database-type &key database-type => result
```

Arguments and Values

database-type	The database type to initialize, i.e. a keyword symbol denoting a known database back-end. Defaults to the value of *default-database-type*.
result	Either NIL if the initialization attempt fails, or t otherwise.

Description

If the back-end specified by *database-type* has not already been initialized, as seen from *initialized-database-types*, an attempt is made to initialize the database. If this attempt succeeds, or the back-end has already been initialized, the function returns t, and places the keyword denoting the database type onto the list stored in *initialized-database-types*, if not already present.

If initialization fails, the function returns NIL, and/or signals an error of type clsql-error. The kind of action taken depends on the back-end and the cause of the problem.

Examples

```
*initialized-database-types*
=> NIL
(setf *default-database-type* :mysql)
=> :MYSQL
(initialize-database-type)
>> Compiling LAMBDA (#:G897 #:G898 #:G901 #:G902):
>> Compiling Top-Level Form:
>>
=> T
*initialized-database-types*
=> (:MYSQL)
(initialize-database-type)
=> T
*initialized-database-types*
=> (:MYSQL)
```

Side Effects

The database back-end corresponding to the database type specified is initialized, unless it has already been initialized. This can involve any number of other side effects, as determined by the back-end implementation (like e.g. loading of foreign code, calling of foreign code, networking operations, etc.). If

initialization is attempted and succeeds, the *database-type* is pushed onto the list stored in *initialized-database-types*.

Affected by

default-database-type
initialized-database-types

Exceptional Situations

If an error is encountered during the initialization attempt, the back-end may signal errors of kind clsql-error.

See Also

initialized-database-types
default-database-type

Notes

None.

Name

RECONNECT — Re-establishes the connection between a database object and its RDBMS.
Function

Syntax

```
reconnect &key database error force => result
```

Arguments and Values

database The database to reconnect, which defaults to the database indicated by *default-database*.

error A boolean flag indicating whether to signal an error if *database* is non-nil but cannot be found. The default value is NIL.

force A Boolean indicating whether to signal an error if the database connection has been lost. The default value is T.

result A Boolean indicating whether the database was successfully reconnected.

Description

Reconnects *database* which defaults to *default-database* to the underlying database management system. On success, T is returned and the variable *default-database* is set to the newly reconnected database. If *database* is a database instance, this object is closed. If *database* is a string, then a connected database whose name matches *database* is sought in the list of connected databases. If no matching database is found and *error* and *database* are both non-NIL an error is signaled, otherwise NIL is returned.

When the current database connection has been lost, if *force* is non-NIL as it is by default, the connection is closed and errors are suppressed. If *force* is NIL and the database connection cannot be closed, an error is signalled.

Examples

```
*default-database*
=> #<CLSQL-SQLITE:SQLITE-DATABASE :memory: OPEN {48CFBEA5}>
(reconnect)
=> #<CLSQL-SQLITE:SQLITE-DATABASE :memory: OPEN {48D64105}>
```

Side Effects

A database connection is re-established and *default-database* may be rebound to the supplied database object.

Affected by

default-database

Exceptional Situations

An error may be signalled if the specified database cannot be located or if the database cannot be closed.

See Also

```
connect
disconnect
disconnect-pooled
```

Notes

None.

Name

STATUS — Print information about connected databases.
Function

Syntax

```
status &optional full =>
```

Arguments and Values

full A boolean indicating whether to print additional table information. The default value is NIL.

Description

Prints information about the currently connected databases to *STANDARD-OUTPUT*. The argument *full* is NIL by default and a value of t means that more detailed information about each database is printed.

Examples

```
(status)

CLSQL STATUS: 2004-06-13 15:07:39
-----------------------------------------------------
   DATABASE                 TYPE               RECORDING
-----------------------------------------------------
   localhost/test/petrov    mysql              nil
   localhost/test/petrov    postgresql         nil
   localhost/test/petrov    postgresql-socket  nil
   test/petrov              odbc               nil
*  :memory:                 sqlite             nil
-----------------------------------------------------

(status t)

CLSQL STATUS: 2004-06-13 15:08:08
-------------------------------------------------------------------------------
   DATABASE                 TYPE               RECORDING  POOLED  TABLES  VIEWS
-------------------------------------------------------------------------------
   localhost/test/petrov    mysql              nil        nil     7       0
   localhost/test/petrov    postgresql         nil        nil     7       0
   localhost/test/petrov    postgresql-socket  nil        nil     7       0
   test/petrov              odbc               nil        nil     7       0
*  :memory:                 sqlite             nil        nil     0       0
-------------------------------------------------------------------------------
```

Side Effects

None.

Affected by

None.

Exceptional Situations

None.

See Also

```
connected-databases
connect
disconnect
*connect-if-exists*
find-database
```

Notes

None.

Name

CREATE-DATABASE — create a database
Function

Syntax

`create-database` *`connection-spec`* `&key` *`database-type`* `=> success`

Arguments and Values

`connection-spec`	A connection specification
`database-type`	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
`success`	A boolean flag. If `T`, a new database was successfully created.

Description

This function creates a database in the database system specified by *`database-type`*.

Examples

```
(create-database '("localhost" "new" "dent" "dent") :database-type :mysql)
=> T

(create-database '("localhost" "new" "dent" "badpasswd") :database-type :mysql)
=>
Error: While trying to access database localhost/new/dent
  using database-type MYSQL:
  Error database-create failed: mysqladmin: connect to server at 'localhost' faile
error: 'Access denied for user: 'root@localhost' (Using password: YES)'
  has occurred.
  [condition type: CLSQL-ACCESS-ERROR]
```

Side Effects

A database will be created on the filesystem of the host.

Exceptional Situations

An exception will be thrown if the database system does not allow new databases to be created or if database creation fails.

See Also

`destroy-database`
`probe-database`
`list-databases`

Notes

This function may invoke the operating systems functions. Thus, some database systems may require the administration functions to be available in the current PATH. At this time, the :mysql backend requires `mysqladmin` and the :postgresql backend requires `createdb`.

`create-database` is a *CLSQL* extension.

Name

DESTROY-DATABASE — destroys a database
Function

Syntax

```
destroy-database connection-spec &key database-type => success
```

Arguments and Values

connection-spec	A connection specification
database-type	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
success	A boolean flag. If T, the database was successfully destroyed.

Description

This function destroys a database in the database system specified by *database-type*.

Examples

```
(destroy-database '("localhost" "new" "dent" "dent") :database-type :postgresql)
=> T

(destroy-database '("localhost" "new" "dent" "dent") :database-type :postgresql)
=>
Error: While trying to access database localhost/test2/root
  using database-type POSTGRESQL:
  Error database-destroy failed: dropdb: database removal failed: ERROR:  database
  has occurred.
  [condition type: CLSQL-ACCESS-ERROR]
```

Side Effects

A database will be removed from the filesystem of the host.

Exceptional Situations

An exception will be thrown if the database system does not allow databases to be removed, the database does not exist, or if database removal fails.

See Also

```
create-database
probe-database
list-databases
```

Notes

This function may invoke the operating systems functions. Thus, some database systems may require the administration functions to be available in the current PATH. At this time, the :mysql backend requires `mysqladmin` and the :postgresql backend requires `dropdb`.

`destroy-database` is a *CLSQL* extension.

Name

PROBE-DATABASE — tests for existence of a database
Function

Syntax

```
probe-database connection-spec &key database-type => success
```

Arguments and Values

connection-spec	A connection specification
database-type	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
success	A boolean flag. If T, the database exists in the database system.

Description

This function tests for the existence of a database in the database system specified by *database-type*.

Examples

```
(probe-database '("localhost" "new" "dent" "dent") :database-type :postgresql)
=> T
```

Side Effects

None

Exceptional Situations

An exception maybe thrown if the database system does not receive administrator-level authentication since function may need to read the administrative database of the database system.

See Also

```
create-database
destroy-database
list-databases
```

Notes

`probe-database` is a *CLSQL* extension.

Name

LIST-DATABASES — List databases matching the supplied connection spec and database type.
Function

Syntax

```
list-databases connection-spec &key database-type => result
```

Arguments and Values

connection-spec	A connection specification
database-type	A database type specifier, i.e. a keyword. This defaults to the value of *default-database-type*
result	A list of matching databases.

Description

This function returns a list of databases existing in the database system specified by *database-type*.

Examples

```
(list-databases '("localhost" "new" "dent" "dent") :database-type :postgresql)
=> ("address-book" "sql-test" "template1" "template0" "test1" "dent" "test")
```

Side Effects

None.

Affected by

None.

Exceptional Situations

An exception maybe thrown if the database system does not receive administrator-level authentication since function may need to read the administrative database of the database system.

See Also

```
create-database
destroy-database
probe-database
```

Notes

`list-databases` is a *CLSQL* extension.

Name

WITH-DATABASE — Execute a body of code with a variable bound to a specified database object.
Macro

Syntax

```
with-database db-var connection-spec &rest connect-args &body body => result
```

Arguments and Values

db-var	A variable which is bound to the specified database.
connection-spec	A vendor specific connection specification supplied as a list or as a string.
connect-args	Other optional arguments to `connect`. This macro use a value of `NIL` for `connect`'s *`make-default`* key, This is in contrast to to the connect function which has a default value of `T` for *`make-default`*.
body	A Lisp code body.
result	Determined by the result of executing the last expression in *`body`*.

Description

Evaluate *`body`* in an environment, where *`db-var`* is bound to the database connection given by *`connection-spec`* and *`connect-args`*. The connection is automatically closed or released to the pool on exit from the body.

Examples

```
(connected-databases)
=> NIL
(with-database (db '(":memory:") :database-type :sqlite
                :make-default nil)
  (database-name db))
=> ":memory:"
(connected-databases)
=> NIL
```

Side Effects

See `connect` and `disconnect`.

Affected by

See `connect` and `disconnect`.

Exceptional Situations

See `connect` and `disconnect`.

See Also

```
connect
disconnect
disconnect-pooled
with-default-database
```

Notes

`with-database` is a *CLSQL* extension.

Name

WITH-DEFAULT-DATABASE — Execute a body of code with *default-database* bound to a specified database.
Macro

Syntax

```
with-default-database database &rest body => result
```

Arguments and Values

database	An active database object.
body	A Lisp code body.
result	Determined by the result of executing the last expression in *body*.

Description

Perform *body* with *DATABASE* bound as *default-database*.

Examples

```
*default-database*
=> #<CLSQL-ODBC:ODBC-DATABASE new/dent OPEN {49095CAD}>

(let ((database (clsql:find-database ":memory:")))
  (with-default-database (database)
    (database-name *default-database*)))
=> ":memory:"
```

Side Effects

None.

Affected by

None.

Exceptional Situations

Calls to *CLSQL* functions in *body* may signal errors if *database* is not an active database object.

See Also

with-database
default-database

Notes

`with-default-database` is a *CLSQL* extension.

The Symbolic SQL Syntax

CLSQL provides a symbolic syntax allowing the construction of SQL expressions as lists delimited by square brackets. The syntax is turned off by default. This section describes utilities for enabling and disabling the square bracket reader syntax and for constructing symbolic SQL expressions.

Tip: just want it on

`file-enable-sql-reader-syntax` at the top of each file is easiest.

Table of Contents

Name

ENABLE-SQL-READER-SYNTAX — Globally enable square bracket reader syntax.
Macro

Syntax

```
enable-sql-reader-syntax =>
```

Arguments and Values

None.

Description

Turns on the SQL reader syntax setting the syntax state such that if the syntax is subsequently disabled, `restore-sql-reader-syntax-state` will enable it again.

Examples

None.

Side Effects

Sets the internal syntax state to enabled.

Modifies the default readtable.

Warning

CLSQL tries to keep track of whether the syntax has already been enabled. This can be problematic if the syntax is somehow disabled externally to *CLSQL* as future attempts to enable the syntax will do nothing--the system thinks it is already enabled. This may happen if there is an enable, but no disable, in a file that is processed with load or compile-file as the lisp implementation will restore the readtable on completion. Or, even if there is a disable but a compiler-error is encountered before running the disable. If you encounter this try running `disable-sql-reader-syntax` a couple times in the REPL.

See `file-enable-sql-reader-syntax` for an alternative.

Affected by

None.

Exceptional Situations

None.

See Also

`disable-sql-reader-syntax`

```
locally-enable-sql-reader-syntax
locally-disable-sql-reader-syntax
restore-sql-reader-syntax-state
file-enable-sql-reader-syntax
```

Notes

The symbolic SQL syntax is disabled by default.

CLSQL differs from CommonSQL in that `enable-sql-reader-syntax` is defined as a macro rather than a function.

Name

DISABLE-SQL-READER-SYNTAX — Globally disable square bracket reader syntax.
Macro

Syntax

```
disable-sql-reader-syntax =>
```

Arguments and Values

None.

Description

Turns off the SQL reader syntax setting the syntax state such that if the syntax is subsequently enabled, `restore-sql-reader-syntax-state` will disable it again.

Examples

None.

Side Effects

Sets the internal syntax state to disabled.

Modifies the default readtable.

Affected by

None.

Exceptional Situations

None.

See Also

`enable-sql-reader-syntax`
`locally-enable-sql-reader-syntax`
`locally-disable-sql-reader-syntax`
`restore-sql-reader-syntax-state`
`file-enable-sql-reader-syntax`

Notes

The symbolic SQL syntax is disabled by default.

CLSQL differs from CommonSQL in that `disable-sql-reader-syntax` is defined as a macro rather than a function.

Name

LOCALLY-ENABLE-SQL-READER-SYNTAX — Locally enable square bracket reader syntax.
Macro

Syntax

```
locally-enable-sql-reader-syntax =>
```

Arguments and Values

None.

Description

Turns on the SQL reader syntax without changing the syntax state such that `restore-sql-reader-syntax-state` will re-establish the current syntax state.

Examples

Intended to be used in a file for code which uses the square bracket syntax without changing the global state.

```
#.(locally-enable-sql-reader-syntax)

... CODE USING SYMBOLIC SQL SYNTAX ...

#.(restore-sql-reader-syntax-state)
```

Side Effects

Modifies the default readtable.

Warning

CLSQL tries to keep track of whether the syntax has already been enabled. This can be problematic if the syntax is somehow disabled externally to *CLSQL* as future attempts to enable the syntax will do nothing--the system thinks it is already enabled. This may happen if there is an enable, but no disable, in a file that is processed with load or compile-file as the lisp implementation will restore the readtable on completion. Or, even if there is a disable but a compiler-error is encountered before running the disable. If you encounter this try running `disable-sql-reader-syntax` a couple times in the REPL.

See `file-enable-sql-reader-syntax` for an alternative.

Affected by

None.

Exceptional Situations

None.

See Also

`enable-sql-reader-syntax`
`disable-sql-reader-syntax`
`locally-disable-sql-reader-syntax`
`restore-sql-reader-syntax-state`
`file-enable-sql-reader-syntax`

Notes

The symbolic SQL syntax is disabled by default.

CLSQL differs from CommonSQL in that `locally-enable-sql-reader-syntax` is defined as a macro rather than a function.

Name

LOCALLY-DISABLE-SQL-READER-SYNTAX — Locally disable square bracket reader syntax.
Macro

Syntax

```
locally-disable-sql-reader-syntax =>
```

Arguments and Values

None.

Description

Turns off the SQL reader syntax without changing the syntax state such that `restore-sql-reader-syntax-state` will re-establish the current syntax state.

Examples

Intended to be used in a file for code in which the square bracket syntax should be disabled without changing the global state.

```
#.(locally-disable-sql-reader-syntax)

... CODE NOT USING SYMBOLIC SQL SYNTAX ...

#.(restore-sql-reader-syntax-state)
```

Side Effects

Modifies the default readtable.

Affected by

None.

Exceptional Situations

None.

See Also

`enable-sql-reader-syntax`
`disable-sql-reader-syntax`
`locally-enable-sql-reader-syntax`
`restore-sql-reader-syntax-state`
`file-enable-sql-reader-syntax`

Notes

The symbolic SQL syntax is disabled by default.

CLSQL differs from CommonSQL in that `locally-disable-sql-reader-syntax` is defined as a macro rather than a function.

Name

RESTORE-SQL-READER-SYNTAX-STATE — Restore square bracket reader syntax to its previous state.
Macro

Syntax

```
restore-sql-reader-syntax-state =>
```

Arguments and Values

None.

Description

Enables the SQL reader syntax if `enable-sql-reader-syntax` has been called more recently than `disable-sql-reader-syntax` and otherwise disables the SQL reader syntax. By default, the SQL reader syntax is disabled.

Examples

See `locally-enable-sql-reader-syntax` and `locally-disable-sql-reader-syntax`.

Side Effects

Reverts the internal syntax state.

Modifies the default readtable.

Affected by

The current internal syntax state.

Exceptional Situations

None.

See Also

`enable-sql-reader-syntax`
`disable-sql-reader-syntax`
`locally-enable-sql-reader-syntax`
`locally-disable-sql-reader-syntax`
`file-enable-sql-reader-syntax`

Notes

The symbolic SQL syntax is disabled by default.

CLSQL differs from CommonSQL in that `restore-sql-reader-syntax-state` is defined as a macro rather than a function.

Name

FILE-ENABLE-SQL-READER-SYNTAX — Enable the square bracket reader syntax for the duration of the file.
Macro

Syntax

```
file-enable-sql-reader-syntax =>
```

Arguments and Values

None.

Description

Uncoditionally enables the SQL reader syntax. Unlike `enable-sql-reader-syntax` and `disable-sql-reader-syntax` which try to keep track of whether the syntax has been enabled or disabled and keep track of the old read-table for restoration this function just enables it unconditionally.

Once enabled this way there is no corresponding disable function but instead relies on being used in a file context. The spec for load [http://www.lispworks.com/documentation/lw51/CLHS/Body/f_load.htm] and compile-file [http://www.lispworks.com/documentation/lw51/CLHS/Body/f_cmp_fi.htm] states that the *readtable* will be restored after processing the file.

Examples

Intended to be used at the top of a file that contains sql reader syntax.

```
(in-package :my-package)
(clsql:file-enable-sql-reader-syntax)
...
;;functions that use the square bracket syntax.
```

Side Effects

Modifies the readtable for #\[and #\]

Affected by

None.

Exceptional Situations

None.

See Also

`enable-sql-reader-syntax`

```
disable-sql-reader-syntax
locally-enable-sql-reader-syntax
locally-disable-sql-reader-syntax
```

Notes

Unique to *CLSQL*, not present in CommonSQL.

Name

SQL — Construct an SQL string from supplied expressions.
Function

Syntax

```
sql &rest args => sql-expression
```

Arguments and Values

args	A set of expressions.
sql-expression	A string representing an SQL expression.

Description

Returns an SQL string generated from the expressions *args*. The expressions are translated into SQL strings and then concatenated with a single space delimiting each expression.

Examples

```
(sql nil)
=> "NULL"

(sql 'foo)
=> "FOO"

(sql "bar")
 => "'bar'"

(sql 10)
=> "10"

(sql '(nil foo "bar" 10))
=> "(NULL,FOO,'bar',10)"

(sql #(nil foo "bar" 10))
=> "NULL,FOO,'bar',10"

(sql [select [foo] [bar] :from [baz]] 'having [= [foo id] [bar id]]
     'and [foo val] '< 5)
=> "SELECT FOO,BAR FROM BAZ HAVING (FOO.ID = BAR.ID) AND FOO.VAL < 5"
```

Side Effects

None.

Affected by

None.

Exceptional Situations

An error of type `sql-user-error` is signalled if any element in *`args`* is not of the supported types (a symbol, string, number or symbolic SQL expression) or a list or vector containing only these supported types.

See Also

`sql-expression`
`sql-operation`
`sql-operator`

Notes

None.

Name

SQL-EXPRESSION — Constructs an SQL expression from supplied keyword arguments.
Function

Syntax

```
sql-expression &key string table alias attribute type => result
```

Arguments and Values

string	A string.
table	A symbol representing a database table identifier.
alias	A table alias.
attribute	A symbol representing an attribute identifier.
type	A type specifier.
result	A object of type sql-expression.

Description

Returns an SQL expression constructed from the supplied arguments which may be combined as follows:

- *attribute* and *type*;
- *attribute*;
- *alias* or *table* and *attribute* and *type*;
- *alias* or *table* and *attribute*;
- *table*, *attribute* and *type*;
- *table* and *attribute*;
- *table* and *alias*;
- *table*;
- *string*.

Examples

```
(sql-expression :table 'foo :attribute 'bar)
=> #<CLSQL-SYS:SQL-IDENT-ATTRIBUTE FOO.BAR>

(sql-expression :attribute 'baz)
=> #<CLSQL-SYS:SQL-IDENT-ATTRIBUTE BAZ>
```

Side Effects

None.

Affected by

None.

Exceptional Situations

An error of type `sql-user-error` is signalled if an unsupported combination of keyword arguments is specified.

See Also

`sql`
`sql-operation`
`sql-operator`

Notes

None.

Name

SQL-OPERATION — Constructs an SQL expression from a supplied operator and arguments.
Function

Syntax

```
sql-operation operator &rest args => result
```

```
sql-operation 'function func &rest args => result
```

Arguments and Values

operator	A symbol denoting an SQL operator.
func	A string denoting an SQL function.
args	A set of arguments for the specified SQL operator or function.
result	A object of type `sql-expression`.

Description

Returns an SQL expression constructed from the supplied SQL operator or function *operator* and its arguments *args*. If *operator* is passed the symbol 'function then the first value in *args* is taken to be a valid SQL function and the remaining values in *args* its arguments.

Examples

```
(sql-operation 'select
          (sql-expression :table 'foo :attribute 'bar)
          (sql-operation 'sum  (sql-expression :table 'foo :attribute 'baz))
          :from
          (sql-expression :table 'foo)
          :where
          (sql-operation '> (sql-expression :attribute 'bar) 12)
          :order-by (sql-operation 'sum (sql-expression :attribute 'baz)))
=> #<SQL-QUERY SELECT FOO.BAR,SUM(FOO.BAZ) FROM FOO WHERE (BAR > 12) ORDER BY SUM(

(sql-operation 'function "strpos" "CLSQL" "SQL")
=> #<CLSQL-SYS:SQL-FUNCTION-EXP STRPOS('CLSQL','SQL')>
```

Side Effects

None.

Affected by

None.

Exceptional Situations

An error of type `sql-user-error` is signalled if *`operator`* is not a symbol representing a supported SQL operator.

See Also

`sql`
`sql-expression`
`sql-operator`

Notes

None.

Name

SQL-OPERATOR — Returns the symbol for the supplied SQL operator.
Function

Syntax

```
sql-operator operator => result
```

Arguments and Values

operator	A symbol denoting an SQL operator.
result	The Lisp symbol used by *CLSQL* to represent the specified operator.

Description

Returns the Lisp symbol corresponding to the SQL operator represented by the symbol *operator*. If *operator* does not represent a supported SQL operator or is not a symbol, nil is returned.

Examples

```
(sql-operator 'like)
=> SQL-LIKE
```

Side Effects

None.

Affected by

None.

Exceptional Situations

None.

See Also

```
sql
sql-expression
sql-operation
```

Notes

CLSQL's symbolic SQL syntax currently has support for the following CommonSQL compatible SQL operators:

```
any
some
all
```

```
not
union
intersect
minus
except
order-by
null
*
+
/
-
like
and
or
in
substr
||
=
<
>
>=
<=
<>
count
max
min
avg
sum
function
between
distinct
nvl
slot-value
userenv
```

as well as the pseudo-operator `function`.

The following operators are provided as *CLSQL* extensions to the CommonSQL API.

```
concat
substring
limit
group-by
having
not-null
exists
uplike
is
==
the
coalesce
view-class
```

Note that some of these operators are not supported by all of the RDBMS supported by *CLSQL* (see the Appendix for details).

Functional Data Definition Language (FDDL)

CLSQL provides a functional DDL which supports the creation and destruction of a variety of database objects including tables, views, indexes and sequences. Functions which return information about currently defined database objects are also provided. In addition, the FDDL includes functionality for examining table attributes and attribute types.

Table of Contents

Name

CREATE-TABLE — Create a database table.
Function

Syntax

```
create-table name description &key database constraints transactions =>
```

Arguments and Values

name	The name of the table as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
description	A list.
constraints	A string, a list of strings or NIL.
transactions	A Boolean. The default value is T.

Description

Creates a table called *name*, which may be a string, symbol or SQL table identifier, in *database* which defaults to *default-database*. *description* is a list whose elements are lists containing the attribute names, types, and other constraints such as not-null or primary-key for each column in the table.

constraints is a string representing an SQL table constraint expression or a list of such strings.

With MySQL databases, if *transactions* is T an InnoDB table is created which supports transactions.

Examples

```
(create-table [foo]
              '(([id] integer)
                ([height] float)
                ([name] (string 24))
                ([comments] text)))
=>
(table-exists-p [foo])
=> T

(create-table [foo] '(([bar] integer :not-null :unique :primary-key)
                      ([baz] string :not-null :unique)))
=>
(table-exists-p [foo])
=> T

(create-table [foo] '(([bar] integer :not-null) ([baz] string :not-null))
              :constraints '("UNIQUE (bar,baz)" "PRIMARY KEY (bar)"))
=>
(table-exists-p [foo])
```

```
=> T
```

Side Effects

A table is created in *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if a relation called *name* already exists.

See Also

```
drop-table
list-tables
table-exists-p
```

Notes

The *constraints* and *transactions* keyword arguments to `create-table` are *CLSQL* extensions. The *transactions* keyword argument is for compatibility with MySQL databases.

Name

DROP-TABLE — Drop a database table.
Function

Syntax

```
drop-table name &key if-does-not-exist database =>
```

Arguments and Values

name	The name of the table as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
if-does-not-exist	A symbol. Meaningful values are :ignore or :error (the default).

Description

Drops the table called *name* from *database* which defaults to *default-database*. If the table does not exist and *if-does-not-exist* is :ignore then `drop-table` returns `NIL` whereas an error is signalled if *if-does-not-exist* is :error.

Examples

```
(table-exists-p [foo])
=> T
(drop-table [foo] :if-does-not-exist :ignore)
=>
(table-exists-p [foo])
=> NIL
```

Side Effects

A table is dropped *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if *name* doesn't exist and *if-does-not-exist* has a value of :error.

See Also

`create-table`

```
list-tables
table-exists-p
```

Notes

The *if-does-not-exist* keyword argument to `drop-table` is a *CLSQL* extension.

Name

LIST-TABLES — Returns a list of database tables.
Function

Syntax

```
list-tables &key owner database => result
```

Arguments and Values

owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A list of strings.

Description

Returns a list of strings representing table names in *database* which defaults to *default-database*. *owner* is NIL by default which means that only tables owned by users are listed. If *owner* is a string denoting a user name, only tables owned by *owner* are listed. If *owner* is :all then all tables are listed.

Examples

```
(list-tables :owner "fred")
=> ("type_table" "type_bigint" "employee" "company" "addr" "ea_join" "big")

(list-tables :owner :all)
=> ("pg_description" "pg_group" "pg_proc" "pg_rewrite" "pg_type" "pg_attribute"
    "pg_class" "pg_inherits" "pg_index" "pg_operator" "pg_opclass" "pg_am"
    "pg_amop" "pg_amproc" "pg_language" "pg_largeobject" "pg_aggregate"
    "pg_trigger" "pg_listener" "pg_cast" "pg_namespace" "pg_shadow"
    "pg_conversion" "pg_depend" "pg_attrdef" "pg_constraint" "pg_database"
    "type_table" "type_bigint" "employee" "company" "pg_statistic" "addr"
    "ea_join" "big")
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
create-table
drop-table
table-exists-p
```

Notes

None.

Name

TABLE-EXISTS-P — Tests for the existence of a database table.
Function

Syntax

```
table-exists-p name &key owner database => result
```

Arguments and Values

name	The name of the table as a string, symbol or SQL expression.
owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A Boolean.

Description

Tests for the existence of an SQL table called *name* in *database* which defaults to *default-database*. *owner* is NIL by default which means that only tables owned by users are examined. If *owner* is a string denoting a user name, only tables owned by *owner* are examined. If *owner* is :all then all tables are examined.

Examples

```
(table-exists-p [foo])
=> T
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
create-table
drop-table
list-tables
```

Notes

None.

Name

CREATE-VIEW — Create a database view.
Function

Syntax

```
create-view name &key as column-list with-check-option database =>
```

Arguments and Values

name	The name of the view as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
as	A symbolic SQL query expression.
column-list	A list.
with-check-option	A Boolean.

Description

Creates a view called *name* in *database* which defaults to *default-database*. The view is created using the query *as* and the columns of the view may be specified using the *column-list* parameter. The *with-check-option* is `NIL` by default but if it has a non-`NIL` value, then all insert/update commands on the view are checked to ensure that the new data satisfy the query *as*.

Examples

```
(create-view [lenins-group]
      :as [select [first-name] [last-name] [email]
                  :from [employee]
    :where [= [managerid] 1]])
=>

(select [*] :from [lenins-group])
=> (("Josef" "Stalin" "stalin@soviet.org")
    ("Leon" "Trotsky" "trotsky@soviet.org")
    ("Nikita" "Kruschev" "kruschev@soviet.org")
    ("Leonid" "Brezhnev" "brezhnev@soviet.org")
    ("Yuri" "Andropov" "andropov@soviet.org")
    ("Konstantin" "Chernenko" "chernenko@soviet.org")
    ("Mikhail" "Gorbachev" "gorbachev@soviet.org")
    ("Boris" "Yeltsin" "yeltsin@soviet.org")
    ("Vladimir" "Putin" "putin@soviet.org")),
   ("first_name" "last_name" "email")
```

Side Effects

A view is created in *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if a relation called *name* already exists.

See Also

`drop-view`
`list-views`
`view-exists-p`

Notes

None.

Name

DROP-VIEW — Drops a database view.
Function

Syntax

```
drop-view name &key if-does-not-exist database =>
```

Arguments and Values

name	The name of the view as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
if-does-not-exist	A symbol. Meaningful values are :ignore or :error (the default).

Description

Drops the view called *name* from *database* which defaults to *default-database*. If the view does not exist and *if-does-not-exist* is :ignore then `drop-view` returns `NIL` whereas an error is signalled if *if-does-not-exist* is :error.

Examples

```
(view-exists-p [foo])
=> T
(drop-view [foo] :if-does-not-exist :ignore)
=>
(view-exists-p [foo])
=> NIL
```

Side Effects

A view is dropped *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if *name* doesn't exist and *if-does-not-exist* has a value of :error.

See Also

`create-view`

```
list-views
view-exists-p
```

Notes

The *if-does-not-exist* keyword argument to `drop-view` is a *CLSQL* extension.

Name

LIST-VIEWS — Returns a list of database views.
Function

Syntax

```
list-views &key owner database => result
```

Arguments and Values

owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A list of strings.

Description

Returns a list of strings representing view names in *database* which defaults to *default-database*. *owner* is NIL by default which means that only views owned by users are listed. If *owner* is a string denoting a user name, only views owned by *owner* are listed. If *owner* is :all then all views are listed.

Examples

```
(list-views :owner "fred")
=> ("lenins_group")

(list-views :owner :all)
=> ("pg_user" "pg_rules" "pg_views" "pg_tables" "pg_indexes" "pg_stats"
    "pg_stat_all_tables" "pg_stat_sys_tables" "pg_stat_user_tables"
    "pg_statio_all_tables" "pg_statio_sys_tables" "pg_statio_user_tables"
    "pg_stat_all_indexes" "pg_stat_sys_indexes" "pg_stat_user_indexes"
    "pg_statio_all_indexes" "pg_statio_sys_indexes" "pg_statio_user_indexes"
    "pg_statio_all_sequences" "pg_statio_sys_sequences"
    "pg_statio_user_sequences" "pg_stat_activity" "pg_stat_database"
    "pg_locks" "pg_settings" "lenins_group")
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

`create-view`
`drop-view`
`view-exists-p`

Notes

`list-views` is a *CLSQL* extension.

Name

VIEW-EXISTS-P — Tests for the existence of a database view.
Function

Syntax

```
view-exists-p name &key owner database => result
```

Arguments and Values

name	The name of the view as a string, symbol or SQL expression.
owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A Boolean.

Description

Tests for the existence of an SQL view called *name* in *database* which defaults to *default-database*. *owner* is NIL by default which means that only views owned by users are examined. If *owner* is a string denoting a user name, only views owned by *owner* are examined. If *owner* is :all then all views are examined.

Examples

```
(view-exists-p [lenins-group])
=> T
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
create-view
drop-view
list-views
```

Notes

`view-exists-p` is a *CLSQL* extension.

Name

CREATE-INDEX — Create a database index.
Function

Syntax

```
create-index name &key on unique attributes database =>
```

Arguments and Values

name	The name of the index as a string, symbol or SQL expression.
on	The name of a table as a string, symbol or SQL expression.
unique	A Boolean.
attributes	A list of attribute names.
database	A database object which defaults to *default-database*.

Description

Creates an index called *name* on the table specified by *on* in *database* which default to *default-database*. The table attributes to use in constructing the index *name* are specified by *attributes*. The *unique* argument is NIL by default but if it has a non-NIL value then the indexed attributes must have unique values.

Examples

```
(create-index [bar] :on [employee]
              :attributes '([first-name] [last-name] [email])
              :unique t)
=>

(index-exists-p [bar])
=> T
```

Side Effects

An index is created in *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if a relation called *name* already exists.

See Also

```
drop-index
list-indexes
index-exists-p
```

Notes

None.

Name

DROP-INDEX — Drop a database index.
Function

Syntax

```
drop-index name &key if-does-not-exist on database =>
```

Arguments and Values

name	The name of the index as a string, symbol or SQL expression.
on	The name of a table as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
if-does-not-exist	A symbol. Meaningful values are :ignore or :error (the default).

Description

Drops the index called *name* in *database* which defaults to *default-database*. If the index does not exist and *if-does-not-exist* is :ignore then *drop-index* returns `NIL` whereas an error is signalled if *if-does-not-exist* is :error.

The argument *on* allows the optional specification of a table to drop the index from. This is required for compatability with MySQL.

Examples

```
(index-exists-p [foo])
=> T
(drop-index [foo] :if-does-not-exist :ignore)
=>
(index-exists-p [foo])
=> NIL
```

Side Effects

An index is dropped in *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if *name* doesn't exist and *if-does-not-exist* has a value of :error.

See Also

`create-index`
`list-indexes`
`index-exists-p`

Notes

The *`if-does-not-exist`* and *`on`* keyword arguments to `drop-index` are *CLSQL* extensions. The keyword argument *`on`* is provided for compatibility with MySQL.

Name

LIST-INDEXES — Returns a list of database indexes.
Function

Syntax

```
list-indexes &key onowner database => result
```

Arguments and Values

owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
on	The name of a table as a string, symbol or SQL expression, a list of such names or NIL.
result	A list of strings.

Description

Returns a list of strings representing index names in *database* which defaults to *default-database*. *owner* is NIL by default which means that only indexes owned by users are listed. If *owner* is a string denoting a user name, only indexes owned by *owner* are listed. If *owner* is :all then all indexes are listed.

The keyword argument *on* limits the results to indexes on the specified tables. Meaningful values for *on* are NIL (the default) which means that all tables are considered, a string, symbol or SQL expression representing a table name in *database* or a list of such table identifiers.

Examples

```
(list-indexes)
=> ("employeepk" "companypk" "addrpk" "bar")

(list-indexes :on '([addr] [company]))
=> ("addrpk" "companypk")
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

`create-index`
`drop-index`
`index-exists-p`

Notes

`list-indexes` is a *CLSQL* extension.

Name

INDEX-EXISTS- — Tests for the existence of a database index.
Function

Syntax

```
index-exists-p name &key owner database => result
```

Arguments and Values

name	The name of the index as a string, symbol or SQL expression.
owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A Boolean.

Description

Tests for the existence of an SQL index called *name* in *database* which defaults to *default-database*. *owner* is NIL by default which means that only indexes owned by users are examined. If *owner* is a string denoting a user name, only indexes owned by *owner* are examined. If *owner* is :all then all indexes are examined.

Examples

```
(index-exists-p [bar])
=> T
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
create-index
drop-index
list-indexes
```

Notes

`index-exists-p` is a *CLSQL* extension.

Name

ATTRIBUTE-TYPE — Returns the type of the supplied attribute.
Function

Syntax

```
attribute-type attribute table &key owner database => type, precision, scale
```

Arguments and Values

attribute	The name of the index as a string, symbol or SQL expression.
table	The name of a table as a string, symbol or SQL expression.
owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
type	A keyword symbol denoting a vendor-specific SQL type.
precision	An integer denoting the precision of the attribute type or NIL.
scale	An integer denoting the scale of the attribute type or NIL.
nulls-accepted	0 or 1.

Description

Returns a keyword symbol representing the vendor-specific field type of the supplied attribute *attribute* in the table specified by *table* in *database* which defaults to *default-database*. *owner* is NIL by default which means that the attribute specified by *attribute*, if it exists, must be user owned else NIL is returned. If *owner* is a string denoting a user name, the attribute, if it exists, must be owned by *owner* else NIL is returned, whereas if *owner* is :all then the attribute, if it exists, will be returned regardless of its owner.

Other information is also returned. The second value is the type precision, the third is the scale and the fourth represents whether or not the attribute accepts null values (a value of 0) or not (a value of 1).

Examples

```
(attribute-type [emplid] [employee])
=> :INT4, 4, NIL, 0
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
list-attributes
list-attribute-types
```

Notes

None.

Name

LIST-ATTRIBUTE-TYPES — Returns information about the attribute types of a table.
Function

Syntax

```
list-attribute-types table &key owner database => result
```

Arguments and Values

table	The name of a table as a string, symbol or SQL expression.
owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A list.

Description

Returns a list containing information about the SQL types of each of the attributes in the table specified by *table* in *database* which has a default value of *default-database*. *owner* is NIL by default which means that only attributes owned by users are listed. If *owner* is a string denoting a user name, only attributes owned by *owner* are listed. If *owner* is :all then all attributes are listed. The elements of the returned list are lists where the first element is the name of the attribute, the second element is its SQL type, the third is the type precision, the fourth is the scale of the attribute and the fifth is 1 if the attribute accepts null values and otherwise 0.

Examples

```
(list-attribute-types [employee])
=> (("emplid" :INT4 4 NIL 0) ("groupid" :INT4 4 NIL 0)
    ("first_name" :VARCHAR 30 NIL 1) ("last_name" :VARCHAR 30 NIL 1)
    ("email" :VARCHAR 100 NIL 1) ("ecompanyid" :INT4 4 NIL 1)
    ("managerid" :INT4 4 NIL 1) ("height" :FLOAT8 8 NIL 1)
    ("married" :BOOL 1 NIL 1) ("birthday" :TIMESTAMP 8 NIL 1)
    ("bd_utime" :INT8 8 NIL 1))
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
attribute-type
list-attribute-types
```

Notes

None.

Name

LIST-ATTRIBUTES — Returns the attributes of a table as a list.
Function

Syntax

```
list-attributes name &key owner database => result
```

Arguments and Values

name	The name of a table as a string, symbol or SQL expression.
owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A list.

Description

Returns a list of strings representing the attributes of table *name* in *database* which defaults to *default-database*. *owner* is NIL by default which means that only attributes owned by users are listed. If *owner* is a string denoting a user name, only attributes owned by *owner* are listed. If *owner* is :all then all attributes are listed.

Examples

```
(list-attributes [employee])
=> ("emplid" "groupid" "first_name" "last_name" "email" "ecompanyid" "managerid"
    "height" "married" "birthday" "bd_utime")
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

attribute-type
list-attribute-types

Notes

None.

Name

CREATE-SEQUENCE — Create a database sequence.
Function

Syntax

```
create-sequence name &key database =>
```

Arguments and Values

name The name of the sequence as a string, symbol or SQL expression.

database A database object which defaults to *default-database*.

Description

Creates a sequence called *name* in *database* which defaults to *default-database*.

Examples

```
(create-sequence [foo])
=>
(sequence-exists-p [foo])
=> T
```

Side Effects

A sequence is created in *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if a relation called *name* already exists.

See Also

```
drop-sequence
list-sequences
sequence-exists-p
sequence-last
sequence-next
set-sequence-position
```

Notes

`create-sequence` is a *CLSQL* extension.

Name

DROP-SEQUENCE — Drop a database sequence.
Function

Syntax

```
drop-sequence name &key if-does-not-exist database =>
```

Arguments and Values

name	The name of the sequence as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
if-does-not-exist	A symbol. Meaningful values are :ignore or :error (the default).

Description

Drops the sequence called *name* from *database* which defaults to *default-database*. If the sequence does not exist and *if-does-not-exist* is :ignore then *drop-sequence* returns NIL whereas an error is signalled if *if-does-not-exist* is :error.

Examples

```
(sequence-exists-p [foo])
=> T
(drop-sequence [foo] :if-does-not-exist :ignore)
=>
(sequence-exists-p [foo])
=> NIL
```

Side Effects

A sequence is dropped from *database*.

Affected by

default-database

Exceptional Situations

An error is signalled if *name* is not a string, symbol or SQL expression. An error of type sql-database-data-error is signalled if *name* doesn't exist and *if-does-not-exist* has a value of :error.

See Also

create-sequence

`list-sequences`
`sequence-exists-p`
`sequence-last`
`sequence-next`
`set-sequence-position`

Notes

`drop-sequence` is a *CLSQL* extension.

Name

LIST-SEQUENCES — Returns a list of database sequences.
Function

Syntax

```
list-sequences &key owner database => result
```

Arguments and Values

owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A list of strings.

Description

Returns a list of strings representing sequence names in *database* which defaults to *default-database*. *owner* is NIL by default which means that only sequences owned by users are listed. If *owner* is a string denoting a user name, only sequences owned by *owner* are listed. If *owner* is :all then all sequences are listed.

Examples

```
(list-sequences)
=> ("foo")
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
create-sequence
drop-sequence
sequence-exists-p
sequence-last
sequence-next
```

`set-sequence-position`

Notes

`list-sequences` is a *CLSQL* extension.

Name

SEQUENCE-EXISTS-P — Tests for the existence of a database sequence.
Function

Syntax

```
sequence-exists-p name &key owner database => result
```

Arguments and Values

name	The name of the sequence as a string, symbol or SQL expression.
owner	A string, NIL or :all.
database	A database object which defaults to *default-database*.
result	A Boolean.

Description

Tests for the existence of an SQL sequence called *name* in *database* which defaults to *default-database*. *owner* is NIL by default which means that only sequences owned by users are examined. If *owner* is a string denoting a user name, only sequences owned by *owner* are examined. If *owner* is :all then all sequences are examined.

Examples

```
(sequence-exists-p [foo])
=> NIL
```

Side Effects

None.

Affected by

default-database

Exceptional Situations

None.

See Also

```
create-sequence
drop-sequence
list-sequences
sequence-last
```

`sequence-next`
`set-sequence-position`

Notes

`sequence-exists-p` is a *CLSQL* extension.

Name

SEQUENCE-LAST — Return the last element in a database sequence.
Function

Syntax

```
sequence-last name &key database => result
```

Arguments and Values

name	The name of the sequence as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
result	An integer.

Description

Return the last value allocated in the sequence called *name* in *database* which defaults to *default-database*.

Examples

```
(sequence-last [foo])
=> 1
```

Side Effects

None.

Affected by

The current value stored in database sequence *name*.

default-database

Exceptional Situations

Will signal an error of type sql-database-data-error if a sequence called *name* does not exist in *database*.

See Also

```
create-sequence
drop-sequence
list-sequences
sequence-exists-p
sequence-next
set-sequence-position
```

Notes

`sequence-last` is a *CLSQL* extension.

Name

SEQUENCE-NEXT — Increment the value of a database sequence.
Function

Syntax

```
sequence-next name &key database => result
```

Arguments and Values

name	The name of the sequence as a string, symbol or SQL expression.
database	A database object which defaults to *default-database*.
result	An integer.

Description

Increment and return the value of the sequence called *name* in *database* which defaults to *default-database*.

Examples

```
(sequence-last [foo])
=> 3
(sequence-next [foo])
=> 4
(sequence-next [foo])
=> 5
(sequence-next [foo])
=> 6
```

Side Effects

Modifies the value of the sequence *name* in *database*.

Affected by

The current value stored in database sequence *name*.

default-database

Exceptional Situations

Will signal an error of type sql-database-data-error if a sequence called *name* does not exist in *database*.

See Also

`create-sequence`

```
drop-sequence
list-sequences
sequence-exists-p
sequence-last
set-sequence-position
```

Notes

`sequence-next` is a *CLSQL* extension.

Name

SET-SEQUENCE-POSITION — Sets the position of a database sequence.
Function

Syntax

```
set-sequence-position name position &key database => result
```

Arguments and Values

name	The name of the sequence as a string, symbol or SQL expression.
position	An integer.
database	A database object which defaults to *default-database*.
result	An integer.

Description

Explicitly set the position of the sequence called *name* in *database*, which defaults to *default-database*, to *position* which is returned.

Examples

```
(sequence-last [foo])
=> 4
(set-sequence-position [foo] 50)
=> 50
(sequence-next [foo])
=> 51
```

Side Effects

Modifies the value of the sequence *name* in *database*.

Affected by

default-database

Exceptional Situations

Will signal an error of type sql-database-data-error if a sequence called *name* does not exist in *database*.

See Also

```
create-sequence
drop-sequence
```

`list-sequences`
`sequence-exists-p`
`sequence-last`
`sequence-next`

Notes

`set-sequence-position` is a *CLSQL* extension.

Name

TRUNCATE-DATABASE — Drop all tables, views, indexes and sequences in a database.
Function

Syntax

```
truncate-database &key database =>
```

Arguments and Values

database A *database object*. This will default to the value of *default-database*.

Description

Drop all tables, views, indexes and sequences in *database* which defaults to *default-database*.

Examples

```
(list-tables)
=> ("type_table" "type_bigint" "employee" "company" "addr" "ea_join" "big")
(list-indexes)
=> ("employeepk" "companypk" "addrpk")
(list-views)
=> ("lenins_group")
(list-sequences)
=> ("foo" "bar")
(truncate-database)
=>
(list-tables)
=> NIL
(list-indexes)
=> NIL
(list-views)
=> NIL
(list-sequences)
=> NIL
```

Side Effects

Modifications are made to the underlying database.

Affected by

None.

Exceptional Situations

Signals an error of type sql-database-error if *database* is not a database object.

See Also

`drop-table`
`drop-view`
`drop-index`
`drop-sequence`

Notes

`truncate-database` is a *CLSQL* extension.

Functional Data Manipulation Language (FDML)

The functional data manipulation interface provided by *CLSQL* includes functions for inserting, updating and deleting records in existing database tables and executing SQL queries and statements with the results of queries returned as Lisp types. SQL statements expressed as strings may be executed with the `query` and `execute-command` functions. The `select` function, on the other hand, allows for the construction of queries in Lisp using the symbolic SQL syntax. Finally, iterative manipulation of query results is supported by `do-query`, `map-query` and an extended clause for the `loop` macro.

Table of Contents

Name

CACHE-TABLE-QUERIES-DEFAULT — Specifies the default behaviour for caching of attribute types.
Variable

Value Type

A valid argument to the *`action`* parameter of `cache-table-queries`, i.e. one of `T`, `NIL`, :flush.

Initial Value

nil

Description

Specifies the default behaivour for caching of attribute types. Meaningful values are `T`, `NIL` and :flush as described for the *`action`* argument to `cache-table-queries`.

Examples

None.

Affected By

None.

See Also

`cache-table-queries`

Notes

None.

Name

CACHE-TABLE-QUERIES — Control the caching of table attribute types.
Function

Syntax

```
cache-table-queries table &key action database =>
```

Arguments and Values

table A string representing a database table, T or :default.

action T, NIL or :flush.

database A *database object*. This will default to the value of *default-database*.

Description

Controls the caching of attribute type information on the table specified by *table* in *database* which defaults to *default-database*. *action* specifies the caching behaviour to adopt. If its value is T then attribute type information is cached whereas if its value is NIL then attribute type information is not cached. If *action* is :flush then all existing type information in the cache for *table* is removed, but caching is still enabled. *table* may be a string representing a table for which the caching action is to be taken while the caching action is applied to all tables if *table* is T. Alternatively, when *table* is :default, the default caching action specified by *cache-table-queries-default* is applied to all tables for which a caching action has not been explicitly set.

Examples

```
(setf *cache-table-queries-default* t)
=> T
(create-table [foo]
              '(([id] integer)
                ([height] float)
                ([name] (string 24))
                ([comments] varchar)))
=>
(cache-table-queries "foo")
=>
(list-attribute-types "foo")
=> (("id" :INT4 4 NIL 1) ("height" :FLOAT8 8 NIL 1) ("name" :BPCHAR 24 NIL 1)
    ("comments" :VARCHAR 255 NIL 1))
(drop-table "foo")
=>
(create-table [foo]
              '(([id] integer)
                ([height] float)
                ([name] (string 36))
                ([comments] (string 100))))
=>
```

```
(cache-table-queries "foo" :action :flush)
=>
(list-attribute-types "foo")
=> (("id" :INT4 4 NIL 1) ("height" :FLOAT8 8 NIL 1) ("name" :BPCHAR 36 NIL 1)
    ("comments" :BPCHAR 100 NIL 1))
```

Side Effects

The internal attribute cache for *`database`* is modified.

Affected by

cache-table-queries-default

Exceptional Situations

None.

See Also

cache-table-queries-default

Notes

None.

Name

INSERT-RECORDS — Insert tuples of data into a database table.
Function

Syntax

```
insert-records &key into attributes values av-pairs query database =>
```

Arguments and Values

into	A string, symbol or symbolic SQL expression representing the name of a table existing in *database*.
attributes	A list of attribute identifiers or NIL.
values	A list of attribute values or NIL.
av-pairs	A list of attribute identifier/value pairs or NIL.
query	A query expression or NIL.
database	A *database object*. This will default to the value of *default-database*.

Description

Inserts records into the table specified by *into* in *database* which defaults to *default-database*.

There are five ways of specifying the values inserted into each row. In the first *values* contains a list of values to insert and *attributes*, *av-pairs* and *query* are NIL. This can be used when values are supplied for all attributes in *into*. In the second, *attributes* is a list of column names, *values* is a corresponding list of values and *av-pairs* and *query* are NIL. In the third, *attributes*, *values* and *query* are NIL and *av-pairs* is an alist of (attribute value) pairs. In the fourth, *values*, *av-pairs* and *attributes* are NIL and *query* is a symbolic SQL query expression in which the selected columns also exist in *into*. In the fifth method, *values* and *av-pairs* are nil and *attributes* is a list of column names and *query* is a symbolic SQL query expression which returns values for the specified columns.

Examples

```
(select [first-name] [last-name] [email]
        :from [employee]
        :where [= [emplid] 11]
        :field-names nil)
=> NIL
(insert-records :into [employee]
                :attributes '(emplid groupid first_name last_name email
                              ecompanyid managerid)
                :values '(11 1 "Yuri" "Gagarin" "gagarin@soviet.org"
                          1 1))
=>
(select [first-name] [last-name] [email]
```

```
        :from [employee]
        :where [= [emplid] 11]
        :field-names nil)
=> (("Yuri" "Gagarin" "gagarin@soviet.org"))
```

Side Effects

Modifications are made to the underlying database.

Affected by

None.

Exceptional Situations

An error of type sql-database-data-error is signalled if *table* is not an existing table in *database* or if the specified attributes are not found.

See Also

```
update-records
delete-records
```

Notes

None.

Name

UPDATE-RECORDS — Updates the values of existing records.
Function

Syntax

```
update-records table &key  attributes values av-pairs where database =>
```

Arguments and Values

table	A string, symbol or symbolic SQL expression representing the name of a table existing in *database*.
attributes	A list of attribute identifiers or NIL.
values	A list of attribute values or NIL.
av-pairs	A list of attribute identifier/value pairs or NIL.
where	A symbolic SQL expression.
database	A *database object*. This will default to the value of *default-database*.

Description

Updates the attribute values of existing records satsifying the SQL expression *where* in the table specified by *table* in *database* which defaults to *default-database*.

There are three ways of specifying the values to update for each row. In the first, *values* contains a list of values to use in the update and *attributes* and *av-pairs* are NIL. This can be used when values are supplied for all attributes in *table*. In the second, *attributes* is a list of column names, *values* is a corresponding list of values and *av-pairs* is NIL. In the third, *attributes* and *values* are NIL and *av-pairs* is an alist of (attribute value) pairs.

Examples

```
(select [first-name] [last-name] [email]
        :from [employee]
        :where [= [emplid] 1]
        :field-names nil)
=> (("Vladimir" "Lenin" "lenin@soviet.org"))
(update-records [employee]
                :av-pairs'((first_name "Yuri")
                           (last_name "Gagarin")
                           (email "gagarin@soviet.org"))
                :where [= [emplid] 1])
=>
(select [first-name] [last-name] [email]
        :from [employee]
        :where [= [emplid] 1]
        :field-names nil)
```

```
=> (("Yuri" "Gagarin" "gagarin@soviet.org"))
```

Side Effects

Modifications are made to the underlying database.

Affected by

None.

Exceptional Situations

An error of type sql-database-data-error is signalled if *table* is not an existing table in *database*, if the specified attributes are not found or if the SQL statement resulting from the symbolic expression *where* does not return a Boolean value.

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

See Also

`insert-records`
`delete-records`

Notes

None.

Name

DELETE-RECORDS — Delete records from a database table.
Function

Syntax

```
delete-records &key from where database =>
```

Arguments and Values

from	A string, symbol or symbolic SQL expression representing the name of a table existing in *database*.
where	A symbolic SQL expression.
database	A *database object*. This will default to the value of *default-database*.

Description

Deletes records satisfying the SQL expression *where* from the table specified by *from* in *database* specifies a database which defaults to *default-database*.

Examples

```
(select [first-name] [last-name] [email]
        :from [employee]
        :where [= [emplid] 11]
        :field-names nil)
=> (("Yuri" "Gagarin" "gagarin@soviet.org"))
(delete-records :from [employee] :where [= [emplid] 11])
=>
(select [first-name] [last-name] [email]
        :from [employee]
        :where [= [emplid] 11]
        :field-names nil)
=> NIL
```

Side Effects

Modifications are made to the underlying database.

Affected by

None.

Exceptional Situations

An error of type sql-database-data-error is signalled if *from* is not an existing table in *database* or if the SQL statement resulting from the symbolic expression *where* does not return a Boolean value.

See Also

```
insert-records
update-records
```

Notes

None.

Name

EXECUTE-COMMAND — Execute an SQL command which returns no values.
Generic Function

Syntax

```
execute-command sql-expression &key database =>
```

Arguments and Values

sql-expression	An *sql expression* that represents an SQL statement which will return no values.
database	A *database object*. This will default to the value of *default-database*.

Description

Executes the SQL command *sql-expression*, which may be a symbolic SQL expression or a string representing any SQL statement apart from a query, on the supplied *database* which defaults to *default-database*.

Examples

```
(execute-command "create table eventlog (time char(30),event char(70))")
=>

(execute-command "create table eventlog (time char(30),event char(70))")
>>
>> While accessing database #<CLSQL-POSTGRESQL:POSTGRESQL-DATABASE {480B2B
>>   with expression "create table eventlog (time char(30),event char(70))
>>   Error NIL: ERROR:  amcreate: eventlog relation already exists
>>   has occurred.
>>
>> Restarts:
>>   0: [ABORT] Return to Top-Level.
>>
>> Debug  (type H for help)
>>
>> (CLSQL-POSTGRESQL::|(PCL::FAST-METHOD DATABASE-EXECUTE-COMMAND (T POSTG
>>  #<unused-arg>
>>  #<unused-arg>
>>  #<unavailable-arg>
>>  #<unavailable-arg>)
>> Source: (ERROR 'SQL-DATABASE-ERROR :DATABASE DATABASE :EXPRESSION ...)
>> 0] 0

(execute-command "drop table eventlog")
=>
```

Side Effects

Whatever effects the execution of the SQL statement has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL statement leads to any errors, an error of type sql-database-error is signalled.

See Also

`query`

Notes

None.

Name

QUERY — Execute an SQL query and return the tuples as a list.
Generic Function

Syntax

```
query query-expression &key database result-types flatp field-names => resul
```

Arguments and Values

query-expression	An *sql expression* that represents an SQL query which is expected to return a (possibly empty) result set.
database	A *database object*. This will default to the value of *default-database*.
flatp	A Boolean whose default value is NIL.
result-types	A *field type specifier*. The default is :auto;.
	The purpose of this argument is cause *CLSQL* to import SQL numeric fields into numeric Lisp objects rather than strings. This reduces the cost of allocating a temporary string and the *CLSQL* users' inconvenience of converting number strings into number objects.
	A value of :auto causes *CLSQL* to automatically convert SQL fields into a numeric format where applicable. The default value of NIL causes all fields to be returned as strings regardless of the SQL type. Otherwise a list is expected which has a element for each field that specifies the conversion. Valid type identifiers are:
	:int Field is imported as a signed integer, from 8-bits to 64-bits depending upon the field type. :double Field is imported as a double-float number. t Field is imported as a string. If the list is shorter than the number of fields, the a value of t is assumed for the field. If the list is longer than the number of fields, the extra elements are ignored.
field-names	A boolean with a default value of T. When T, this function returns a second value of a list of field names. When NIL, this function only returns one value - the list of rows.
result	A list representing the result set obtained. For each tuple in the result set, there is an element in this list, which is itself a list of all the attribute values in the tuple.

Description

Executes the SQL query expression *query-expression*, which may be an SQL expression or a string, on the supplied *database* which defaults to *default-database*. *result-types* is a list of symbols which specifies the lisp type for each field returned by *query-expression*.

If *result-types* is NIL all results are returned as strings whereas the default value of :auto means that the lisp types are automatically computed for each field.

field-names is T by default which means that the second value returned is a list of strings representing the columns selected by *query-expression*. If *field-names* is NIL, the list of column names is not returned as a second value.

flatp has a default value of NIL which means that the results are returned as a list of lists.If FLATP is T and only one result is returned for each record selected by *query-expression*, the results are returned as elements of a list.

Examples

```
(query "select emplid,first_name,last_name,height from employee where emplid = 1")
=> ((1 "Vladimir" "Lenin" 1.5564661d0)),
   ("emplid" "first_name" "last_name" "height")

(query "select emplid,first_name,last_name,height from employee where emplid = 1"
       :field-names nil)
=> ((1 "Vladimir" "Lenin" 1.5564661d0))

(query "select emplid,first_name,last_name,height from employee where emplid = 1"
       :field-names nil
       :result-types nil)
=> (("1" "Vladimir" "Lenin" "1.5564661"))

(query "select emplid,first_name,last_name,height from employee where emplid = 1"
       :field-names nil
       :result-types '(:int t t :double))
=> ((1 "Vladimir" "Lenin" 1.5564661))

(query "select last_name from employee where emplid > 5" :flatp t)
=> ("Andropov" "Chernenko" "Gorbachev" "Yeltsin" "Putin"),
   ("last_name")

(query "select last_name from employee where emplid > 10"
       :flatp t
       :field-names nil)
=> NIL
```

Side Effects

Whatever effects the execution of the SQL query has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

See Also

execute-command

```
print-query
do-query
map-query
loop
select
```

Notes

The *`field-names`* and *`result-types`* keyword arguments are a *CLSQL* extension.

Name

PRINT-QUERY — Prints a tabular report of query results.
Function

Syntax

```
print-query query-expression &key titles formats sizes stream database =>
```

Arguments and Values

query-expression	An *sql expression* that represents an SQL query which is expected to return a (possibly empty) result set.
database	A *database object*. This will default to the value of *default-database*.
titles	A list of strings or `NIL` which is the default value.
formats	A list of strings, `NIL` or `T` which is the default value.
sizes	A list of numbers, `NIL` or `T` which is the default value.
stream	An output stream or `T` which is the default value.

Description

Prints a tabular report of the results returned by the SQL query *query-expression*, which may be a symbolic SQL expression or a string, in *database* which defaults to *default-database*. The report is printed onto *stream* which has a default value of `T` which means that *standard-output* is used. The *title* argument, which defaults to `NIL`, allows the specification of a list of strings to use as column titles in the tabular output. *sizes* accepts a list of column sizes, one for each column selected by *query-expression*, to use in formatting the tabular report. The default value of `T` means that minimum sizes are computed. *formats* is a list of format strings to be used for printing each column selected by *query-expression*. The default value of *formats* is `T` meaning that ~A is used to format all columns or ~VA if column sizes are used.

Examples

```
(print-query [select [emplid] [first-name] [last-name] [email]
                     :from [employee]
                     :where [< [emplid] 5]]
              :titles '("ID" "FORENAME" "SURNAME" "EMAIL"))
ID FORENAME SURNAME  EMAIL
1  Vladimir Lenin    lenin@soviet.org
2  Josef    Stalin   stalin@soviet.org
3  Leon     Trotsky  trotsky@soviet.org
4  Nikita   Kruschev kruschev@soviet.org
=>

(print-query "select emplid,first_name,last_name,email from employee where emplid
             :titles '("ID" "FORENAME" "SURNAME" "EMAIL"))
ID FORENAME   SURNAME   EMAIL
```

```
5  Leonid     Brezhnev  brezhnev@soviet.org
6  Yuri       Andropov  andropov@soviet.org
7  Konstantin Chernenko chernenko@soviet.org
8  Mikhail    Gorbachev gorbachev@soviet.org
9  Boris      Yeltsin   yeltsin@soviet.org
10 Vladimir   Putin     putin@soviet.org
=>
```

Side Effects

None.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

See Also

```
query
do-query
map-query
loop
select
```

Notes

None.

Name

SELECT — Executes a query given the supplied constraints.
Function

Syntax

```
select &rest identifiers &key all distinct from group-by having limit offset
```

Arguments and Values

identifiers	A set of *sql expressions* each of which indicates a column to query.
all	A Boolean.
distinct	A Boolean.
from	One or more SQL expression representing tables.
group-by	An SQL expression.
having	An SQL expression.
limit	A non-negative integer.
offset	A non-negative integer.
order-by	An SQL expression.
set-operation	An SQL expression.
where	An SQL expression.
database	A *database object*. This will default to the value of *default-database*.
flatp	A Boolean whose default value is `NIL`.
result-types	A *field type specifier*. The default is :auto.

The purpose of this argument is cause *CLSQL* to import SQL numeric fields into numeric Lisp objects rather than strings. This reduces the cost of allocating a temporary string and the *CLSQL* users' inconvenience of converting number strings into number objects.

A value of :auto causes *CLSQL* to automatically convert SQL fields into a numeric format where applicable. The default value of `NIL` causes all fields to be returned as strings regardless of the SQL type. Otherwise a list is expected which has a element for each field that specifies the conversion. Valid type identifiers are:

:int Field is imported as a signed integer, from 8-bits to 64-bits depending upon the field type.
:double Field is imported as a double-float number.
t Field is imported as a string.
If the list is shorter than the number of fields, the a value of t is assumed for the field.
If the list is longer than the number of fields, the extra elements are ignored.

field-names A boolean with a default value of T. When T, this function returns a second value of a list of field names. When NIL, this function only returns one value - the list of rows.

refresh This value is only considered when CLOS objects are being selected. A boolean with a default value of NIL. When the value of the caching keyword is T, a second equivalent select call will return the same view class instance objects. When refresh is T, then slots of the existing instances are updated as necessary. In such cases, you may wish to override the hook instance-refresh.

caching This value is only considered when CLOS objects are being selected. A boolean with a default value of *default-caching*. *CLSQL* caches objects in accordance with the CommonSQL interface: a second equivalent select call will return the same view class instance objects.

result A list representing the result set obtained. For each tuple in the result set, there is an element in this list, which is itself a list of all the attribute values in the tuple.

Description

Executes a query on *database*, which has a default value of *default-database*, specified by the SQL expressions supplied using the remaining arguments in *args*. The select function can be used to generate queries in both functional and object oriented contexts.

In the functional case, the required arguments specify the columns selected by the query and may be symbolic SQL expressions or strings representing attribute identifiers. Type modified identifiers indicate that the values selected from the specified column are converted to the specified lisp type. The keyword arguments *all*, *distinct*, *from*, *group-by*, *having*, *limit*, *offset*, *order-by*, *set-operation* and *where* are used to specify, using the symbolic SQL syntax, the corresponding components of the SQL query generated by the call to select.

result-types is a list of symbols which specifies the lisp type for each field returned by the query. If *result-types* is NIL all results are returned as strings whereas the default value of :auto means that the lisp types are automatically computed for each field. *field-names* is T by default which means that the second value returned is a list of strings representing the columns selected by the query. If *field-names* is NIL, the list of column names is not returned as a second value.

In the object oriented case, the required arguments to select are symbols denoting View Classes which specify the database tables to query. In this case, select returns a list of View Class instances whose slots are set from the attribute values of the records in the specified table. Slot-value is a legal operator which can be employed as part of the symbolic SQL syntax used in the *where* keyword argument to select. *refresh* is NIL by default which means that the View Class instances returned are retrieved from a cache if an equivalent call to select has previously been issued. If *refresh* is true, the View Class instances returned are updated as necessary from the database and the generic function instance-refreshed is called to perform any necessary operations on the updated instances.

In both object oriented and functional contexts, *flatp* has a default value of NIL which means that the results are returned as a list of lists. If *flatp* is t and only one result is returned for each record selected in the query, the results are returned as elements of a list.

Examples

```
(select [first-name] :from [employee] :flatp t :distinct t
                     :field-names nil
```

```
                    :result-types nil
                    :order-by [first-name])
=> ("Boris" "Josef" "Konstantin" "Leon" "Leonid" "Mikhail" "Nikita" "Vladimir"
    "Yuri")

(select [first-name] [count [*]] :from [employee]
                         :result-types nil
            :group-by [first-name]
            :order-by [first-name]
            :field-names nil)
=> (("Boris" "1") ("Josef" "1") ("Konstantin" "1") ("Leon" "1") ("Leonid" "1")
    ("Mikhail" "1") ("Nikita" "1") ("Vladimir" "2") ("Yuri" "1"))

(select [last-name] :from [employee]
                    :where [like [email] "%org"]
      :order-by [last-name]
      :field-names nil
      :result-types nil
      :flatp t)
=> ("Andropov" "Brezhnev" "Chernenko" "Gorbachev" "Kruschev" "Lenin" "Putin"
    "Stalin" "Trotsky" "Yeltsin")

(select [max [emplid]] :from [employee]
                       :flatp t
                :field-names nil
                       :result-types :auto)
=> (10)

(select [avg [height]] :from [employee] :flatp t :field-names nil)
=> (1.58999584d0)

(select [emplid] [last-name] :from [employee] :where [= [emplid] 1])
=> ((1 "Lenin")),
   ("emplid" "last_name")

(select [emplid :string] :from [employee]
                         :where [= 1 [emplid]]
                         :field-names nil
                         :flatp t)
=> ("1")

(select [emplid] :from [employee] :order-by [emplid]
                 :where [not [between [* [emplid] 10] [* 5 10] [* 10 10]]]
                 :field-names nil
                 :flatp t)
=> (1 2 3 4)

(select [emplid] :from [employee]
        :where [in [emplid] '(1 2 3 4)]
        :flatp t
        :order-by [emplid]
        :field-names nil)
=> (1 2 3 4)
```

```
(select [emplid] :from [employee]
        :order-by [emplid]
        :limit 5
        :offset 3
        :field-names nil
        :flatp t)
=> (4 5 6 7 8)

(select [first-name] [last-name] :from [employee]
        :field-names nil
        :order-by '(([first-name] :asc) ([last-name] :desc)))
=> (("Boris" "Yeltsin") ("Josef" "Stalin") ("Konstantin" "Chernenko")
    ("Leon" "Trotsky") ("Leonid" "Brezhnev") ("Mikhail" "Gorbachev")
    ("Nikita" "Kruschev") ("Vladimir" "Putin") ("Vladimir" "Lenin")
    ("Yuri" "Andropov"))

(select [last-name] :from [employee]
                :set-operation [union [select [first-name] :from [employee]
                                              :order-by [last-name]]]
                :flatp t
                :result-types nil
                :field-names nil)
=> ("Andropov" "Boris" "Brezhnev" "Chernenko" "Gorbachev" "Josef" "Konstantin"
    "Kruschev" "Lenin" "Leon" "Leonid" "Mikhail" "Nikita" "Putin" "Stalin"
    "Trotsky" "Vladimir" "Yeltsin" "Yuri")
```

Side Effects

Whatever effects the execution of the SQL query has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

See Also

```
query
print-query
do-query
map-query
loop
instance-refreshed
```

Notes

The `select` function is actually implemented in *CLSQL* with a single &rest parameter (which is subsequently destructured) rather than the keyword parameters presented here for the purposes of exposition.

This means that incorrect or missing keywords or values may not trigger errors in the way that they would if `select` had been defined using keyword arguments.

The *`field-names`* and *`result-types`* keyword arguments are a *CLSQL* extension.

`select` is common across the functional and object-oriented data manipulation languages.

Name

DO-QUERY — Iterate over all the tuples of a query.
Macro

Syntax

```
do-query ((&rest args) query-expression &key database result-types &body bod
```

Arguments and Values

args	A list of variable names.
query-expression	An *sql expression* that represents an SQL query which is expected to return a (possibly empty) result set, where each tuple has as many attributes as *function* takes arguments.
database	A *database object*. This will default to *default-database*.
result-types	A *field type specifier*. The default is `NIL`. See `query` for the semantics of this argument.
body	A body of Lisp code, like in a `destructuring-bind` form.
result	The result of executing *body*.

Description

Repeatedly executes *body* within a binding of *args* on the fields of each row selected by the SQL query *query-expression*, which may be a string or a symbolic SQL expression, in *database* which defaults to *default-database*.

The body of code is executed in a block named nil which may be returned from prematurely via `return` or `return-from`. In this case the result of evaluating the `do-query` form will be the one supplied to `return` or `return-from`. Otherwise the result will be nil.

The body of code appears also is if wrapped in a `destructuring-bind` form, thus allowing declarations at the start of the body, especially those pertaining to the bindings of the variables named in *args*.

result-types is a list of symbols which specifies the lisp type for each field returned by *query-expression*. If *result-types* is `NIL` all results are returned as strings whereas the default value of :auto means that the lisp types are automatically computed for each field.

query-expression may be an object query (i.e., the selection arguments refer to View Classes), in which case *args* are bound to the tuples of View Class instances returned by the object oriented query.

Examples

```
(do-query ((salary name) "select salary,name from simple")
  (format t "~30A gets $~2,5$~%" name (read-from-string salary)))
>> Mai, Pierre                    gets $10000.00
>> Hacker, Random J.              gets $08000.50
```

```
=> NIL

(do-query ((salary name) "select salary,name from simple")
  (return (cons salary name)))
=> ("10000.00" . "Mai, Pierre")

(let ((result '()))
  (do-query ((name) [select [last-name] :from [employee]
                            :order-by [last-name]])
    (push name result))
  result)
=> ("Yeltsin" "Trotsky" "Stalin" "Putin" "Lenin" "Kruschev" "Gorbachev"
    "Chernenko" "Brezhnev" "Andropov")

(let ((result '()))
  (do-query ((e) [select 'employee :order-by [last-name]])
    (push (slot-value e 'last-name) result))
  result)
=> ("Yeltsin" "Trotsky" "Stalin" "Putin" "Lenin" "Kruschev" "Gorbachev"
    "Chernenko" "Brezhnev" "Andropov")
```

Side Effects

Whatever effects the execution of the SQL query has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

If the number of variable names in *args* and the number of attributes in the tuples in the result set don't match up, an error is signalled.

See Also

query
map-query
print-query
loop
select

Notes

The *result-types* keyword argument is a *CLSQL* extension.

do-query is common across the functional and object-oriented data manipulation languages.

Name

LOOP — Extension to Common Lisp `Loop` to iterate over all the tuples of a query via a loop clause.
Loop Clause

Syntax

`{as | for} var [type-spec] being {each | the} {record | records | tuple | tuples}`

Arguments and Values

var A `d-var-spec`, as defined in the grammar for `loop`-clauses in the ANSI Standard for Common Lisp. This allows for the usual loop-style destructuring.

type-spec An optional `type-spec` either simple or destructured, as defined in the grammar for `loop`-clauses in the ANSI Standard for Common Lisp.

query An *sql expression* that represents an SQL query which is expected to return a (possibly empty) result set, where each tuple has as many attributes as *function* takes arguments.

database An optional *database object*. This will default to the value of *default-database*.

Description

This clause is an iteration driver for `loop`, that binds the given variable (possibly destructured) to the consecutive tuples (which are represented as lists of attribute values) in the result set returned by executing the SQL *query* expression on the *database* specified.

query may be an object query (i.e., the selection arguments refer to View Classes), in which case the supplied variable is bound to the tuples of View Class instances returned by the object oriented query.

Examples

```
(defvar *my-db* (connect '("dent" "newesim" "dent" "dent"))
"My database"
=> *MY-DB*
(loop with time-graph = (make-hash-table :test #'equal)
   with event-graph = (make-hash-table :test #'equal)
   for (time event) being the tuples of "select time,event from log"
   from *my-db*
   do
     (incf (gethash time time-graph 0))
     (incf (gethash event event-graph 0))
   finally
     (flet ((show-graph (k v) (format t "~40A => ~5D~%" k v)))
       (format t "~&Time-Graph:~%==========~%")
       (maphash #'show-graph time-graph)
       (format t "~&~%Event-Graph:~%===========~%")
       (maphash #'show-graph event-graph))
     (return (values time-graph event-graph)))
>> Time-Graph:
>> ==========
>> D                                        => 53000
```

```
>> X                                        =>     3
>> test-me                                  =>  3000
>>
>> Event-Graph:
>> ============
>> CLOS Benchmark entry.                    =>  9000
>> Demo Text...                             =>     3
>> doit-text                                =>  3000
>> C    Benchmark entry.                    => 12000
>> CLOS Benchmark entry                     => 32000
=> #<EQUAL hash table, 3 entries {48350A1D}>
=> #<EQUAL hash table, 5 entries {48350FCD}>

(loop for (forename surname)
      being each tuple in
        [select [first-name] [last-name] :from [employee]
                :order-by [last-name]]
      collect (concatenate 'string forename " " surname))
=> ("Yuri Andropov" "Leonid Brezhnev" "Konstantin Chernenko" "Mikhail Gorbachev"
    "Nikita Kruschev" "Vladimir Lenin" "Vladimir Putin" "Josef Stalin"
    "Leon Trotsky" "Boris Yeltsin")

(loop for (e) being the records in
     [select 'employee :where [< [emplid] 4] :order-by [emplid]]
  collect (slot-value e 'last-name))
=> ("Lenin" "Stalin" "Trotsky")
```

Side Effects

Whatever effects the execution of the SQL query has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

Otherwise, any of the exceptional situations of `loop` applies.

See Also

```
query
map-query
do-query
print-query
select
```

Notes

The *`database`* loop keyword is a *CLSQL* extension.

The extended `loop` syntax is common across the functional and object-oriented data manipulation languages.

Name

MAP-QUERY — Map a function over all the tuples from a query
Function

Syntax

```
map-query output-type-spec function query-expression &key database result-types =>
```

Arguments and Values

output-type-spec	A sequence type specifier or nil.
function	A function designator. *function* takes a single argument which is the atom value for a query single with a single column or is a list of values for a multi-column query.
query-expression	An *sql expression* that represents an SQL query which is expected to return a (possibly empty) result set.
database	A *database object*. This will default to the value of *default-database*.
result-types	A *field type specifier*. The default is NIL. See query for the semantics of this argument.
result	If *output-type-spec* is a type specifier other than nil, then a sequence of the type it denotes. Otherwise nil is returned.

Description

Applies *function* to the successive tuples in the result set returned by executing the SQL *query-expression*. If the *output-type-spec* is nil, then the result of each application of *function* is discarded, and map-query returns nil. Otherwise the result of each successive application of *function* is collected in a sequence of type *output-type-spec*, where the jths element is the result of applying *function* to the attributes of the jths tuple in the result set. The collected sequence is the result of the call to map-query.

If the *output-type-spec* is a subtype of list, the result will be a list.

If the *result-type* is a subtype of vector, then if the implementation can determine the element type specified for the *result-type*, the element type of the resulting array is the result of *upgrading* that element type; or, if the implementation can determine that the element type is unspecified (or *), the element type of the resulting array is t; otherwise, an error is signaled.

If *result-types* is NIL all results are returned as strings whereas the default value of :auto means that the lisp types are automatically computed for each field.

query-expression may be an object query (i.e., the selection arguments refer to View Classes), in which case the supplied function is applied to the tuples of View Class instances returned by the object oriented query.

Examples

```
(map-query 'list #'(lambda (tuple)
                     (multiple-value-bind (salary name) tuple
                        (declare (ignorable name))
                        (read-from-string salary)))
            "select salary,name from simple where salary > 8000")
=> (10000.0 8000.5)

(map-query '(vector double-float)
           #'(lambda (tuple)
               (multiple-value-bind (salary name) tuple
                  (declare (ignorable name))
                  (let ((*read-default-float-format* 'double-float))
                    (coerce (read-from-string salary) 'double-float))
           "select salary,name from simple where salary > 8000")))
=> #(10000.0d0 8000.5d0)
(type-of *)
=> (SIMPLE-ARRAY DOUBLE-FLOAT (2))

(let (list)
  (values (map-query nil #'(lambda (tuple)
                             (multiple-value-bind (salary name) tuple
                               (push (cons name (read-from-string salary)) list))
                           "select salary,name from simple where salary > 8000"))
          list))
=> NIL
=> (("Hacker, Random J." . 8000.5) ("Mai, Pierre" . 10000.0))

(map-query 'vector #'identity
           [select [last-name] :from [employee] :flatp t
                   :order-by [last-name]])
=> #("Andropov" "Brezhnev" "Chernenko" "Gorbachev" "Kruschev" "Lenin" "Putin"
     "Stalin" "Trotsky" "Yeltsin")

(map-query 'list #'identity
           [select [first-name] [last-name] :from [employee]
                   :order-by [last-name]])
=> (("Yuri" "Andropov") ("Leonid" "Brezhnev") ("Konstantin" "Chernenko")
    ("Mikhail" "Gorbachev") ("Nikita" "Kruschev") ("Vladimir" "Lenin")
    ("Vladimir" "Putin") ("Josef" "Stalin") ("Leon" "Trotsky")
    ("Boris" "Yeltsin"))

(map-query 'list #'last-name [select 'employee :order-by [emplid]])
=> ("Lenin" "Stalin" "Trotsky" "Kruschev" "Brezhnev" "Andropov" "Chernenko"
    "Gorbachev" "Yeltsin" "Putin")
```

Side Effects

Whatever effects the execution of the SQL query has on the underlying database, if any.

Affected by

None.

Exceptional Situations

If the execution of the SQL query leads to any errors, an error of type sql-database-error is signalled.

An error of type type-error must be signaled if the *output-type-spec* is not a recognizable subtype of list, not a recognizable subtype of vector, and not nil.

An error of type type-error should be signaled if *output-type-spec* specifies the number of elements and the size of the result set is different from that number.

See Also

query
do-query
print-query
loop
select

Notes

The *result-types* keyword argument is a *CLSQL* extension.

map-query is common across the functional and object-oriented data manipulation languages.

Transaction Handling

This section describes the interface provided by *CLSQL* for handling database transactions. The interface allows for opening transaction blocks, committing or rolling back changes made and controlling autocommit behaviour.

Note

In contrast to CommonSQL, *CLSQL*, by default, starts in transaction AUTOCOMMIT mode (see `set-autocommit`). To begin a transaction in autocommit mode, `start-transaction` has to be called explicitly.

Table of Contents

Name

START-TRANSACTION — Open a transaction block.
Function

Syntax

```
start-transaction &key database => NIL
```

Arguments and Values

database A *database object*. This will default to the value of *default-database*.

Description

Starts a transaction block on *database* which defaults to *default-database* and which continues until `rollback` or `commit` are called.

Examples

```
(in-transaction-p)
=> NIL
(select [*] :from [foo] :field-names nil)
=> NIL
(start-transaction)
=> NIL
(in-transaction-p)
=> T
(insert-records :into [foo] :av-pairs '(([bar] 1) ([baz] "one")))
=>
(select [*] :from [foo] :field-names nil)
=> ((1 "one"))
(rollback)
=> NIL
(in-transaction-p)
=> NIL
(select [*] :from [foo] :field-names nil)
=> NIL
```

Side Effects

Autocommit mode is disabled and if *database* is currently within the scope of a transaction, all commit and rollback hooks are removed and the transaction level associated with *database* is modified.

Affected by

None.

Exceptional Situations

Signals an error of type sql-database-error if *database* is not a database object.

See Also

`commit`
`rollback`
`in-transaction-p`
`set-autocommit`
`with-transaction`

Notes

`start-transaction` is a *CLSQL* extension.

Name

COMMIT — Commit modifications made in the current transaction.
Function

Syntax

```
commit &key database => NIL
```

Arguments and Values

`database` A *database object*. This will default to the value of *default-database*.

Description

If `database`, which defaults to *default-database*, is currently within the scope of a transaction, commits changes made since the transaction began.

Examples

```
(in-transaction-p)
=> NIL
(select [*] :from [foo] :field-names nil)
=> NIL
(start-transaction)
=> NIL
(in-transaction-p)
=> T
(insert-records :into [foo] :av-pairs '(([bar] 1) ([baz] "one")))
=>
(select [*] :from [foo] :field-names nil)
=> ((1 "one"))
(commit)
=> NIL
(in-transaction-p)
=> NIL
(select [*] :from [foo] :field-names nil)
=> ((1 "one"))
```

Side Effects

Changes made within the scope of the current transaction are committed in the underlying database and the transaction level of `database` is reset.

Affected by

The transaction level of `database` which indicates whether a transaction has been initiated by a call to `start-transaction` since the last call to `rollback` or `commit`.

Exceptional Situations

Signals an error of type sql-database-error if *database* is not a database object. A warning of type sql-warning is signalled if there is no transaction in progress.

See Also

```
start-transaction
rollback
in-transaction-p
add-transaction-commit-hook
set-autocommit
with-transaction
```

Notes

None.

Name

ROLLBACK — Roll back modifications made in the current transaction.
Function

Syntax

```
rollback &key database => NIL
```

Arguments and Values

`database` A *database object*. This will default to the value of *default-database*.

Description

If `database`, which defaults to *default-database*, is currently within the scope of a transaction, rolls back changes made since the transaction began.

Examples

```
(in-transaction-p)
=> NIL
(select [*] :from [foo] :field-names nil)
=> NIL
(start-transaction)
=> NIL
(in-transaction-p)
=> T
(insert-records :into [foo] :av-pairs '(([bar] 1) ([baz] "one")))
=>
(select [*] :from [foo] :field-names nil)
=> ((1 "one"))
(rollback)
=> NIL
(in-transaction-p)
=> NIL
(select [*] :from [foo] :field-names nil)
=> NIL
```

Side Effects

Changes made within the scope of the current transaction are reverted in the underlying database and the transaction level of `database` is reset.

Affected by

The transaction level of `database` which indicates whether a transaction has been initiated by a call to `start-transaction` since the last call to `rollback` or `commit`.

Exceptional Situations

Signals an error of type sql-database-error if *`database`* is not a database object. A warning of type sql-warning is signalled if there is no transaction in progress.

See Also

```
start-transaction
commit
in-transaction-p
add-transaction-rollback-hook
set-autocommit
with-transaction
```

Notes

None.

Name

IN-TRANSACTION-P — A predicate for testing whether a transaction is currently in progress.
Function

Syntax

```
in-transaction-p &key database => result
```

Arguments and Values

database A *database object*. This will default to the value of *default-database*.

result A Boolean.

Description

A predicate to test whether *database*, which defaults to *default-database*, is currently within the scope of a transaction.

Examples

```
(in-transaction-p)
=> NIL
(start-transaction)
=> NIL
(in-transaction-p)
=> T
(commit)
=> NIL
(in-transaction-p)
=> NIL
```

Side Effects

None.

Affected by

None.

Exceptional Situations

None.

See Also

start-transaction

```
commit
rollback
set-autocommit
```

Notes

`in-transaction-p` is a *CLSQL* extension.

Name

ADD-TRANSACTION-COMMIT-HOOK — Specify hooks to be run when committing changes.
Function

Syntax

```
add-transaction-commit-hook commit-hook &key database => result
```

Arguments and Values

commit-hook	A designator for a function with no required arguments.
database	A *database object*. This will default to the value of *default-database*.
result	The list of currently defined commit hooks for *database*.

Description

Adds *commit-hook*, which should a designator for a function with no required arguments, to the list of hooks run when `commit` is called on *database* which defaults to *default-database*.

Examples

```
(start-transaction)
=> NIL
(add-transaction-commit-hook #'(lambda () (print "Successfully committed.")))
=> (#<Interpreted Function (LAMBDA # #) {48E2E689}>)
(commit)
"Successfully committed."
=> NIL
```

Side Effects

commit-hook is added to the list of commit hooks for *database*.

Affected by

None.

Exceptional Situations

If *commit-hook* has one or more required arguments, an error will be signalled when `commit` is called.

See Also

`commit`
`rollback`

`add-transaction-rollback-hook`
`with-transaction`

Notes

`add-transaction-commit-hook` is a *CLSQL* extension.

Name

ADD-TRANSACTION-ROLLBACK-HOOK — Specify hooks to be run when rolling back changes.
Function

Syntax

```
add-transaction-rollback-hook rollback-hook &key database => result
```

Arguments and Values

rollback-hook	A designator for a function with no required arguments.
database	A *database object*. This will default to the value of *default-database*.
result	The list of currently defined rollback hooks for *database*.

Description

Adds *rollback-hook*, which should a designator for a function with no required arguments, to the list of hooks run when `rollback` is called on *database* which defaults to *default-database*.

Examples

```
(start-transaction)
=> NIL
(add-transaction-rollback-hook #'(lambda () (print "Successfully rolled back.")))
=> (#<Interpreted Function (LAMBDA # #) {48E37C31}>)
(rollback)
"Successfully rolled back."
=> NIL
```

Side Effects

rollback-hook is added to the list of rollback hooks for *database*.

Affected by

None.

Exceptional Situations

If *rollback-hook* has one or more required arguments, an error will be signalled when `rollback` is called.

See Also

`commit`

`rollback`
`add-transaction-commit-hook`

Notes

`add-transaction-rollback-hook` is a *CLSQL* extension.

Name

SET-AUTOCOMMIT — Turn on or off autocommit for a database.
Function

Syntax

```
set-autocommit value &key database => result
```

Arguments and Values

value	A Boolean specifying the desired autocommit behaviour for *database*.
database	A *database object*. This will default to the value of *default-database*.
result	The previous autocommit value for *database*.

Description

Turns autocommit off for *database* if *value* is NIL, and otherwise turns it on. Returns the old value of autocommit flag.

For RDBMS (such as Oracle) which don't automatically commit changes, turning autocommit on has the effect of explicitly committing changes made whenever SQL statements are executed.

Autocommit is turned on by default.

Examples

Side Effects

database is associated with the specified autocommit mode.

Affected by

None.

Exceptional Situations

None.

See Also

```
start-transaction
commit
add-transaction-commit-hook
with-transaction
```

Notes

`set-autocommit` is a *CLSQL* extension.

Name

WITH-TRANSACTION — Execute a body of code within a transaction.
Macro

Syntax

```
with-transaction &key database &rest body => result
```

Arguments and Values

database A *database object*. This will default to the value of *default-database*.

body A body of Lisp code.

result The result of executing *body*.

Description

Starts a transaction in the database specified by *database*, which is *default-database* by default, and executes *body* within that transaction. If *body* aborts or throws, *database* is rolled back and otherwise the transaction is committed.

Examples

```
(in-transaction-p)
=> NIL
(select [email] :from [employee] :where [= [emplid] 1] :flatp t :field-names nil)
=> ("lenin@soviet.org")
(with-transaction ()
   (update-records [employee]
                   :av-pairs '((email "lenin-nospam@soviet.org"))
                   :where [= [emplid] 1]))
=> NIL
(select [email] :from [employee] :where [= [emplid] 1] :flatp t :field-names nil)
=> ("lenin-nospam@soviet.org")
(in-transaction-p)
=> NIL
```

Side Effects

Changes specified in *body* may be made to the underlying database if *body* completes successfully.

Affected by

None.

Exceptional Situations

Signals an error of type sql-database-error if *database* is not a database object.

See Also

```
start-transaction
commit
rollback
add-transaction-commit-hook
add-transaction-rollback-hook
```

Notes

None.

Object Oriented Data Definition Language (OODDL)

The Object Oriented Data Definition Language (OODDL) provides access to relational SQL tables using Common Lisp Object System (CLOS) objects. SQL tables are mapped to CLOS objects with the SQL columns being mapped to slots of the CLOS object.

The mapping between SQL tables and CLOS objects is defined with the macro `def-view-class`. SQL tables are created with `create-view-from-class` and SQL tables can be deleted with `drop-view-from-class`.

Note

The above functions refer to the Lisp *view* of the SQL table. This Lisp view should not be confused with SQL `VIEW` statement.

Table of Contents

Name

STANDARD-DB-OBJECT — Superclass for all *CLSQL* View Classes.
Class

Class Precedence List

standard-db-object, standard-object, t

Description

This class is the superclass of all *CLSQL* View Classes.

Class details

```
(defclass STANDARD-DB-OBJECT ()(...))
```

Slots

slot VIEW-DATABASE is of type (OR NULL DATABASE) which stores the associated database for the instance.

Name

DEFAULT-STRING-LENGTH — Default length of SQL strings.
Variable

Value Type

Fixnum

Initial Value

255

Description

If a slot of a class defined by `def-view-class` is of the type *`string`* or *`varchar`* and does not have a length specified, then the value of this variable is used as SQL length.

Examples

```
(let ((*default-string-length* 80))
  (def-view-class s80 ()
    ((a :type string)
     (b :type (string 80))
     (c :type varchar))))
=> #<Standard-Db-Class S80 {480A431D}>

(create-view-from-class 's80)
=>
(table-exists-p [s80])
=> T
```

The above code causes a SQL table to be created with the SQL command

```
CREATE TABLE (A VARCHAR(80), B CHAR(80), C VARCHAR(80))
```

Affected By

Some SQL backends do not support *`varchar`* lengths greater than 255.

See Also

None.

Notes

This is a CLSQL extension to the CommonSQL API.

Name

CREATE-VIEW-FROM-CLASS — Create a SQL table from a *View Class*.
Function

Syntax

```
create-view-from-class view-class-name &key database transactions =>
```

Arguments and Values

view-class-name	The name of a *View Class* that has been defined with `def-view-class`.
database	The *database* in which to create the SQL table. This will default to the value of *default-database*.
transactions	When `NIL` specifies that a table type which does not support transactions should be used.

Description

Creates a table as defined by the *View Class* *view-class-name* in *database*.

Examples

```
(def-view-class foo () ((a :type (string 80))))
=> #<Standard-Db-Class FOO {4807F7CD}>
(create-view-from-class 'foo)
=>
(list-tables)
=> ("FOO")
```

Side Effects

Causes a table to be created in the SQL database.

Affected by

Most SQL database systems will signal an error if a table creation is attempted when a table with the same name already exists. The SQL user, as specified in the database connection, must have sufficient permission for table creation.

Exceptional Situations

A condition will be signaled if the table can not be created in the SQL database.

See Also

`def-view-class`

```
drop-view-from-class
```

Notes

Currently, only MySQL supports transactionless tables. *CLSQL* provides the ability to create such tables for applications which would benefit from faster table access and do not require transaction support.

The case of the table name is determined by the type of the database. MySQL, for example, creates databases in upper-case while PostgreSQL uses lowercase.

Name

DEF-VIEW-CLASS — Defines CLOS classes with mapping to SQL database.
Macro

Syntax

```
def-view-class name superclasses slots &rest class-options => class
```

Arguments and Values

name	The class name.
superclasses	The superclasses for the defined class.
slots	The class slot definitions.
class options	The class options.
class	The defined class.

Slot Options

- *:db-kind* - specifies the kind of database mapping which is performed for this slot and defaults to *:base* which indicates that the slot maps to an ordinary column of the database table. A *:db-kind* value of *:key* indicates that this slot is a special kind of *:base* slot which maps onto a column which is one of the unique keys for the database table, the value *:join* indicates this slot represents a join onto another *View Class* which contains View Class objects, and the value *:virtual* indicates a standard CLOS slot which does not map onto columns of the database table.
- *:db-info* - if a slot is specified with *:db-kind :join*, the slot option *:db-info* contains a property list which specifies the nature of the join. The valid members of the list are:
  - *:join-class* *class-name* - the name of the class to join on.
  - *:home-key* *slot-name* - the name of the slot of this class for joining
  - *:foreign-key* *slot-name* - the name of the slot of the *:join-class* for joining
  - *:target-slot* *target-slot* - this is an optional parameter. If specified, then the join slot of the defining class will contain instances of this target slot rather than of the join class. This can be useful when the *:join-class* is an intermediate class in a *many-to-many* relationship and the application is actually interested in the *:target-slot*.
  - *:retrieval* *time* - The default value is *:deferred*, which defers filling this slot until the value is accessed. The other valid value is *:immediate* which performs the SQL query when the instance of the class is created. In this case, the *:set* is automatically set to NIL
  - *:set* *set* - This controls what is stored in the join slot. The default value is T. When *set* is T and *target-slot* is undefined, the join slot will contain a list of instances of the join class. Whereas, if *target-slot* is defined, then the join slot will contain a list of pairs of *(target-value join-instance)*. When *set* is NIL, the join slot will contain a single instances.
- *:type* - for slots of *:db-kind :base* or *:key*, the *:type* slot option has a special interpretation such that Lisp types, such as string, integer and float are automatically converted into appropriate SQL

types for the column onto which the slot maps. This behaviour may be overridden using the *:db-type* slot option. The valid values are:

string - a variable length character field up to *default-string-length* characters.
(string n) - a fixed length character field *n* characters long.
varchar - a variable length character field up to *default-string-length* characters.
(varchar n) - a variable length character field up to *n* characters in length.
char - a single character field
integer - signed integer at least 32-bits wide
(integer n)
float
(float n)
long-float
number
(number n)
(number n p)
tinyint - An integer column 8-bits wide. [not supported by all database backends]
smallint - An integer column 16-bits wide. [not supported by all database backends]
bigint - An integer column 64-bits wide. [not supported by all database backends]
universal-time - an integer field sufficiently wide to store a universal-time. On most databases, a slot of this type assigned a SQL type of *BIGINT*
wall-time - a slot which stores a date and time in a SQL timestamp column. *CLSQL* provides a number of time manipulation functions to support objects of type wall-time.
date - a slot which stores the date (without any time of day resolution) in a column. *CLSQL* provides a number of time manipulation functions that operate on date values.
duration - stores a duration structure. *CLSQL* provides routines for wall-time and duration processing.
boolean - stores a T or NIL value.
generalized-boolean - similar to a *boolean* in that either a T or NIL value is stored in the SQL database. However, any Lisp object can be stored in the Lisp object. A Lisp value of NIL is stored as FALSE in the database, any other Lisp value is stored as TRUE.
keyword - stores a keyword
symbol - stores a symbol
list - stores a list by writing it to a string. The items in the list must be able to be readable written.
vector - stores a vector similarly to *list*
array - stores a array similarly to *list*

- *:column* - specifies the name of the SQL column which the slot maps onto, if *:db-kind* is not *:virtual*, and defaults to the slot name. If the slot name is used for the SQL column name, any hypens in the slot name are converted to underscore characters.

- *:void-value* - specifies the value to store in the Lisp instance if the SQL value is NULL and defaults to NIL.

- *:db-constraints* - is a keyword symbol representing an SQL column constraint expression or a list of such symbols. The following column constraints are supported: :not-null, :primary-key, :unique, :unsigned (MySQL specific), :zerofill (MySQL specific) and :auto-increment (MySQL specific).

- *:db-type* - a string to specify the SQL column type. If specified, this string overrides the SQL column type as computed from the *:type* slot value.

- *:db-reader* - If a string, then when reading values from the database, the string will be used for a format string, with the only value being the value from the database. The resulting string will be used as the slot value. If a function then it will take one argument, the value from the database, and return the value that should be put into the slot. If a symbol, then the symbol-function of the symbol will be used.

- *:db-writer* - If a string, then when reading values from the slot for the database, the string will be used for a format string, with the only value being the value of the slot. The resulting string will be used as the column value in the database. If a function then it will take one argument, the value of the slot, and return the value that should be put into the database. If a symbol, then the symbol-function of the symbol will be used.

Class Options

- *:base-table* - specifies the name of the SQL database table. The default value is the class name. Like slot names, hypens in the class name are converted to underscore characters.
- *:normalizedp* - specifies whether this class uses normalized inheritance from parent classes. Defaults to nil, i.e. non-normalized schemas. When true, SQL database tables that map to this class and parent classes are joined on their primary keys to get the full set of database columns for this class. This means that the primary key of the base class will be copied to all subclasses as we insert so that all parent classes of an instance will have the same value in their primary key slots (see tests/ds-nodes.lisp and oodml.lisp)

Description

Creates a *View Class* called *name* whose slots *slots* can map onto the attributes of a table in a database. If *superclasses* is NIL then the superclass of *class* will be *standard-db-object*, otherwise *superclasses* is a list of superclasses for *class* which must include *standard-db-object* or a descendent of this class.

Normalized inheritance schemas

Specifying that :normalizedp is T tells *CLSQL* to normalize the database schema for inheritance. What this means is shown in the examples below.

With :normalizedp equal to NIL (the default) the class inheritance would result in the following:

```
(def-view-class node ()
  ((title :accessor title :initarg :title :type (varchar 240))))

SQL table NODE:
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| TITLE | varchar(240) | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+

(def-view-class user (node)
  ((user-id :accessor user-id :initarg :user-id
            :type integer :db-kind :key :db-constraints (:not-null))
   (nick :accessor nick :initarg :nick :type (varchar 64))))

SQL table USER:
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| USER_ID | int(11)      | NO   | PRI |         |       |
| NICK    | varchar(64)  | YES  |     | NULL    |       |
```

```
| TITLE   | varchar(240) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+
```

Using :normalizedp T, both view-classes need a primary key to join them on:

```
(def-view-class node ()
  ((node-id :accessor node-id :initarg :node-id
            :type integer :db-kind :key
            :db-constraints (:not-null))
   (title :accessor title :initarg :title :type (varchar 240))))

SQL table NODE:
+---------+--------------+------+-----+---------+-------+
| Field   | Type         | Null | Key | Default | Extra |
+---------+--------------+------+-----+---------+-------+
| NODE_ID | int(11)      | NO   | PRI |         |       |
| TITLE   | varchar(240) | YES  |     | NULL    |       |
+---------+--------------+------+-----+---------+-------+

(def-view-class user (node)
  ((user-id :accessor user-id :initarg :user-id
            :type integer :db-kind :key :db-constraints (:not-null))
   (nick :accessor nick :initarg :nick :type (varchar 64)))
  (:normalizedp t))

SQL table USER:
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| USER_ID | int(11)     | NO   | PRI |         |       |
| NICK    | varchar(64) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

In this second case, all slots of the view-class 'node are also available in view-class 'user, and can be used as one would expect. For example, with the above normalized view-classes 'node and 'user, and SQL tracing turned on:

```
CLSQL> (setq test-user (make-instance 'user :node-id 1 :nick "test-user"
                                            :title "This is a test user"))
#<USER {1003B392E1}>

CLSQL> (update-records-from-instance test-user :database db)

;; .. => INSERT INTO NODE (NODE_ID,TITLE) VALUES (1,'This is a test user')
;; .. <= T
;; .. => INSERT INTO USER (USER_ID,NICK) VALUES (1,'test-user')
;; .. <= T
1


CLSQL> (node-id test-user)
```

```
1

CLSQL> (title test-user)
"This is a test user"

CLSQL> (nick test-user)
"test-user"
```

Notes from a refactor of this code. There are many assumptions that need to be met for normalized classes to work * The each of the classes should have its own single key column (of a different name) that will contain an identical value. EG: node has a node-id, setting which is a node has a node-id and a setting-id which must be equal. You cannot use node-id as the primary key on both tables (as I would have expected). The exception to this seems to be if your class has no slots at all, then you dont need to have a single key column, because your class is fully represented in the db by its parent(s) * more than one parent class per normalized class should be considered experimental and untested (vaya con Dios) * There are a few code paths that just dont pay any attention to normalized classes eg: delete-records-for-instance

Examples

The following examples are from the *CLSQL* test suite.

```
(def-view-class person (thing)
  ((height :db-kind :base :accessor height :type float
           :initarg :height)
   (married :db-kind :base :accessor married :type boolean
            :initarg :married)
   (birthday :type clsql:wall-time :initarg :birthday)
   (bd-utime :type clsql:universal-time :initarg :bd-utime)
   (hobby :db-kind :virtual :initarg :hobby :initform nil)))

(def-view-class employee (person)
  ((emplid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :emplid)
   (groupid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :groupid)
   (first-name
    :accessor first-name
    :type (varchar 30)
    :initarg :first-name)
   (last-name
    :accessor last-name
    :type (varchar 30)
    :initarg :last-name)
   (email
    :accessor employee-email
    :type (varchar 100)
```

```
     :initarg :email)
    (ecompanyid
     :type integer
     :initarg :companyid)
    (company
     :accessor employee-company
     :db-kind :join
     :db-info (:join-class company
      :home-key ecompanyid
      :foreign-key companyid
      :set nil))
    (managerid
     :type integer
     :initarg :managerid)
    (manager
     :accessor employee-manager
     :db-kind :join
     :db-info (:join-class employee
      :home-key managerid
      :foreign-key emplid
      :set nil))
    (addresses
     :accessor employee-addresses
     :db-kind :join
     :db-info (:join-class employee-address
      :home-key emplid
      :foreign-key aemplid
      :target-slot address
      :set t)))
  (:base-table employee))

(def-view-class company ()
  ((companyid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :companyid)
   (groupid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :groupid)
   (name
    :type (varchar 100)
    :initarg :name)
   (presidentid
    :type integer
    :initarg :presidentid)
   (president
    :reader president
    :db-kind :join
    :db-info (:join-class employee
     :home-key presidentid
     :foreign-key emplid
```

```
      :set nil))
    (employees
     :reader company-employees
     :db-kind :join
     :db-info (:join-class employee
      :home-key (companyid groupid)
      :foreign-key (ecompanyid groupid)
      :set t))))

(def-view-class address ()
  ((addressid
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :addressid)
   (street-number
    :type integer
    :initarg :street-number)
   (street-name
    :type (varchar 30)
    :void-value ""
    :initarg :street-name)
   (city
    :column "city_field"
    :void-value "no city"
    :type (varchar 30)
    :initarg :city)
   (postal-code
    :column zip
    :type integer
    :void-value 0
    :initarg :postal-code))
  (:base-table addr))

;; many employees can reside at many addressess
(def-view-class employee-address ()
  ((aemplid :type integer :initarg :emplid)
   (aaddressid :type integer :initarg :addressid)
   (verified :type boolean :initarg :verified)
   (address :db-kind :join
     :db-info (:join-class address
      :home-key aaddressid
      :foreign-key addressid
      :retrieval :immediate)))
  (:base-table "ea_join"))

(def-view-class deferred-employee-address ()
  ((aemplid :type integer :initarg :emplid)
   (aaddressid :type integer :initarg :addressid)
   (verified :type boolean :initarg :verified)
   (address :db-kind :join
     :db-info (:join-class address
      :home-key aaddressid
      :foreign-key addressid
```

```
      :retrieval :deferred
      :set nil)))
  (:base-table "ea_join"))
```

Side Effects

Creates a new CLOS class.

Affected by

Nothing.

Exceptional Situations

None.

See Also

`create-view-from-class`
`standard-db-object`
`drop-view-from-class`

Notes

The actual SQL type for a column depends up the database type in which the SQL table is stored. As an example, the view class type *`(varchar 100)`* specifies a SQL column type *`VARCHAR(100)`* in MySQL and a column type *`VARCHAR2(100)`* in Oracle

The actual lisp type for a slot may be different than the value specified by the *`:type`* attribute. For example, a slot declared with "*`:type (string 30)`*" actually sets the slots Lisp type as *`(or null string)`*. This is to allow a `NIL` value or a string shorter than 30 characters to be stored in the slot.

Name

DROP-VIEW-FROM-CLASS — Delete table from SQL database.
Function

Syntax

```
drop-view-from-class view-class-name &key database =>
```

Arguments and Values

view-class-name	The name of the *View Class*.
database	*database object*. This will default to the value of *default-database*.

Description

Removes a table defined by the *View Class* *view-class-name* from *database* which defaults to **default-database**.

Examples

```
(list-tables)
=> ("FOO" "BAR")
(drop-view-from-class 'foo)
=>
(list-tables)
=> ("BAR")
```

Side Effects

Deletes a table from the SQL database.

Affected by

Whether the specified table exists in the SQL database.

Exceptional Situations

A condition may be signalled if the table does not exist in the SQL database or if the SQL connection does not have sufficient permissions to delete tables.

See Also

```
create-view-from-class
def-view-class
```

Notes

None.

Name

LIST-CLASSES — List classes for tables in SQL database.
Function

Syntax

```
list-classes &key test root-class database => classes
```

Arguments and Values

test	a function used to filter the search. By default, *identity* is used which will return all classes.
root-class	specifies the root class to the search. By default, *standard-db-object* is used which is the root for all view classes.
database	The *database* to search for view classes. This will default to the value of *default-database*.
classes	List of view classes.

Description

Returns a list of all the View Classes which have been defined in the Lisp session and are connected to *database* and which descended from the class *root-class* and which satisfy the function *test*.

Examples

```
(list-classes)
=> (#<clsql-sys::standard-db-class big> #<clsql-sys::standard-db-class employee-ad
    #<clsql-sys::standard-db-class address> #<clsql-sys::standard-db-class company
    #<clsql-sys::standard-db-class employee>)

(list-classes :test #'(lambda (c) (> (length (symbol-name (class-name c))) 3)))
=> (#<clsql-sys::standard-db-class employee-address> #<clsql-sys::standard-db-clas
    #<clsql-sys::standard-db-class company> #<clsql-sys::standard-db-class employe
```

Side Effects

None.

Affected by

Which view classes have been defined in the Lisp session.

Exceptional Situations

None.

See Also

`def-view-class`

Notes

None.

Object Oriented Data Manipulation Language (OODML)

Object Oriented Data Manipulation Language (OODML) provides a Common Lisp Object System (CLOS) interface to SQL databases. View classes are defined with the OODDL interface and objects are read and written with the OODML.

The main function for reading data with the OODML is the `select` function. The `select` is also used in the FDML. However, when `select` is given a view class name, it returns a list of instances of view classes.

View class instances can be updated to reflect any changes in the database with the functions `update-slot-from-record` and `update-instance-from-records`.

To update the database to reflect changes made to instances of view classes, use the functions `update-records-from-instance`, `update-record-from-slot`, and `update-record-from-slots`.

The function `delete-instance-records` deletes the records corresponding to an instance of a view class.

Table of Contents

Name

DB-AUTO-SYNC — Enables SQL storage during Lisp object creation.
Variable

Value Type

Boolean

Initial Value

`NIL`

Description

When this variable is `T` an instance is stored in the SQL database when the instance is created by `make-instance`. Furthermore, the appropriate database records are updated whenever the slots of a *View Class* instance are modified.

When this variable is `NIL`, which is the default value, *CLSQL* behaves like CommonSQL: instances of view classes are stored or updated in the SQL database only when `update-record-from-instance`, `update-record-from-slot` or `update-record-from-slots` are called.

Examples

```
(let ((instance (make-instance 'foo)))
  (update-records-from-instance instance))

;; is equivalent to

(let ((*db-auto-sync* t))
  (make-instance 'foo))

      ;; and

      (progn
        (setf (slot-value instance 'bar) "baz")
        (update-record-from-slot instance 'bar))

      ;; is equivalent to

      (let ((*db-auto-sync* t))
        (setf (slot-value instance 'bar) "baz"))
```

Affected By

None.

See Also

`update-records-from-instance`

```
update-record-from-slot
update-record-from-slots
```

Notes

This is a CLSQL extension to the CommonSQL API.

Name

DEFAULT-CACHING — Controls the default caching behavior.
Variable

Value Type

Boolean

Initial Value

`T`

Description

This variable stores the default value of the `CACHING` keyword for the `select`.

Examples

```
(let ((*default-caching* nil)))
  (select 'foo))

;; is equivalent to

(select 'foo :caching nil)
```

Affected By

None.

See Also

`select`

Notes

This is a CLSQL extension to the CommonSQL API. CommonSQL has caching on at all times.

Name

DEFAULT-UPDATE-OBJECTS-MAX-LEN — The default maximum number of objects each query to perform a join
Variable

Value Type

(or null integer)

Initial Value

`NIL`

Description

This special variable provides the default value for the *`max-len`* argument of the function `update-object-joins`.

Examples

```
(setq *default-update-objects-max-len* 100)
```

Affected By

None.

See Also

`update-object-joins`

Notes

None.

Name

INSTANCE-REFRESHED — User hook to call on object refresh.
Generic function

Syntax

```
instance-refreshed object =>
```

Arguments and Values

object The *View Class* object which is being refreshed.

Description

Provides a hook which is called within an object oriented call to `select` with a non-nil value of *refresh* when the *View Class* instance *object* has been updated from the database. A method specialised on standard-db-object is provided which has no effects. Methods specialised on particular View Classes can be used to specify any operations that need to be made on View Classes instances which have been updated in calls to `select`.

Examples

```
(slot-value employee1 'email)
=> "lenin@soviet.org"
(defmethod instance-refreshed ((e employee))
   (format t "~&Details for ~A ~A have been updated from the database."
           (slot-value e 'first-name)
           (slot-value e 'last-name)))
=> #<Standard-Method INSTANCE-REFRESHED (EMPLOYEE) {48174D9D}>
(select 'employee :where [= [slot-value 'employee 'emplid] 1] :flatp t)
=> (#<EMPLOYEE {48149995}>)
(slot-value (car *) 'email)
=> "lenin@soviet.org"
(update-records [employee] :av-pairs '(([email] "v.lenin@soviet.org"))
               :where [= [emplid] 1])
=>
(select 'employee :where [= [slot-value 'employee 'emplid] 1] :flatp t)
=> (#<EMPLOYEE {48149995}>)
(slot-value (car *) 'email)
=> "lenin@soviet.org"
(select 'employee :where [= [slot-value 'employee 'emplid] 1] :flatp t :refresh t)
Details for Vladimir Lenin have been updated from the database.
=> (#<EMPLOYEE {48149995}>)
(slot-value (car *) 'email)
=> "v.lenin@soviet.org"
```

Side Effects

The user hook function may cause side effects.

Exceptional Situations

None.

See Also

`select`

Notes

None.

Name

DELETE-INSTANCE-RECORDS — Delete SQL records represented by a *View Class* object.
Function

Syntax

```
delete-instance-records object =>
```

Arguments and Values

object An instance of a *View Class*.

Description

Deletes the records represented by *object* in the appropriate table of the database associated with *object*. If *object* is not yet associated with a database, an error is signalled.

Examples

```
(def-view-class tab ()
  ((a :initarg :a :type integer :db-kind :key)
   (b :initarg :b :type string)))
=> #<Standard-Db-Class TAB {49B01845}>
(create-view-from-class 'tab)
=>
(defvar obj (let ((*db-auto-sync* t))
              (make-instance 'tab :a 5 :b "the string")))
=> OBJ
(start-sql-recording :type :both)
=>
(delete-instance-records obj)
;; 2004-07-17 11:07:19 foo/bar/baz => DELETE FROM tab WHERE tab.a = 5
;; 2004-07-17 11:07:19 foo/bar/baz <= T
=>
```

Side Effects

Deletes data from the SQL database.

Affected by

Permissions granted by the SQL database to the user in the database connection.

Exceptional Situations

An exception may be signaled if the database connection user does not have sufficient privileges to modify the database. An error of type sql-database-error is signalled if *object* is not associated with an active database.

See Also

```
update-records
delete-records
update-records-from-instance
```

Notes

Instances are referenced in the database by values stored in the key slots. If `delete-records-from-instance` is called with an instance of a class that does not contain any keys, then all records in that table will be deleted.

Name

UPDATE-RECORDS-FROM-INSTANCE — Update database from view class object.
Function

Syntax

```
update-records-from-instance object &key database =>
```

Arguments and Values

object An instance of a *View Class*.

database *database object*. This will default to the value of *default-database*.

Description

Using an instance of a *View Class*, *object*, update the table that stores its instance data. *database* specifies the database in which the update is made only if *object* is not associated with a database. In this case, a record is created in the appropriate table of *database* using values from the slot values of *object*, and *object* becomes associated with *database*.

Examples

```
(select [email] :from [employee] :where [= [emplid] 1] :field-names nil :flatp t)
=> ("lenin@soviet.org")
(defvar *e1* (car (select 'employee :where [= [slot-value 'employee 'emplid] 1] :f
=> *E1*
(slot-value *e1* 'email)
=> "lenin@soviet.org"
(setf (slot-value *e1* 'email) "v.lenin@soviet.org")
=> "v.lenin@soviet.org"
(update-records-from-instance *e1*)
=>
(select [email] :from [employee] :where [= [emplid] 1] :field-names nil :flatp t)
=> ("v.lenin@soviet.org")
```

Side Effects

Modifies the database.

Affected by

Nothing.

Exceptional Situations

Database errors.

See Also

```
update-record-from-slot
update-record-from-slots
update-records
```

Notes

None.

Name

UPDATE-RECORD-FROM-SLOT — Updates database from slot value.
Function

Syntax

```
update-record-from-slot object slot &key database =>
```

Arguments and Values

object	An instance of a *View Class*.
slot	The name of a slot in *object*.
database	A *database object*. This will default to the value of *default-database*.

Description

Updates the value stored in the column represented by the slot, specified by the CLOS slot name *slot*, of *View Class* instance *object*. *database* specifies the database in which the update is made only if *object* is not associated with a database. In this case, a record is created in *database* and the attribute represented by *slot* is initialised from the value of the supplied slots with other attributes having default values. Furthermore, *object* becomes associated with *database*.

Examples

```
(select [email] :from [employee] :where [= [emplid] 1] :field-names nil :flatp t)
=> ("lenin@soviet.org")
(defvar *e1* (car (select 'employee :where [= [slot-value 'employee 'emplid] 1] :f
=> *E1*
(slot-value *e1* 'email)
=> "lenin@soviet.org"
(setf (slot-value *e1* 'email) "v.lenin@soviet.org")
=> "v.lenin@soviet.org"
(update-record-from-slot *e1* 'email)
=>
(select [email] :from [employee] :where [= [emplid] 1] :field-names nil :flatp t)
=> ("v.lenin@soviet.org")
```

Side Effects

Modifies database.

Affected By

Nothing.

Exceptional Situations

Database errors.

See Also

```
update-record-from-slots
update-records-from-instance
```

Notes

None.

Name

UPDATE-RECORD-FROM-SLOTS — Update database from slots of view class object.
function

syntax

```
update-record-from-slots object slots &key database =>
```

Arguments and Values

object An instance of a *View Class*.

slots A list of slot names in *object*.

database A *database object*. This will default to the value of *default-database*.

Description

Updates the values stored in the columns represented by the slots, specified by the clos slot names *slots*, of *View Class* instance *object*. *database* specifies the database in which the update is made only if *object* is not associated with a database. In this case, a record is created in the appropriate table of *database* and the attributes represented by *slots* are initialised from the values of the supplied slots with other attributes having default values. Furthermore, *object* becomes associated with *database*.

Examples

```
(select [last-name] [email] :from [employee] :where [= [emplid] 1] :field-names ni
=> (("Lenin" "lenin@soviet.org"))
(defvar *e1* (car (select 'employee :where [= [slot-value 'employee 'emplid] 1] :f
=> *E1*
(slot-value *e1* 'last-name)
=> "Lenin"
(slot-value *e1* 'email)
=> "lenin@soviet.org"
(setf (slot-value *e1* 'last-name) "Ivanovich")
=> "Ivanovich"
(setf (slot-value *e1* 'email) "v.ivanovich@soviet.org")
=> "v.ivanovich@soviet.org"
(update-record-from-slots *e1* '(email last-name))
=>
(select [last-name] [email] :from [employee] :where [= [emplid] 1] :field-names ni
=> (("Ivanovich" "v.ivanovich@soviet.org"))
```

Side Effects

Modifies the SQL database.

Affected by

Nothing.

Exceptional Situations

Database errors.

See Also

```
update-record-from-slot
update-records-from-instance
```

Notes

None.

Name

UPDATE-INSTANCE-FROM-RECORDS — Update slot values from database.
Function

Syntax

```
update-instance-from-records object &key database => object
```

Arguments and Values

object An instance of a *View Class*.

database A *database object*. This will default to the value of *default-database*.

Description

Updates the slot values of the *View Class* instance *object* using the attribute values of the appropriate table of *database* which defaults to the database associated with *object* or, if *object* is not associated with a database, *default-database*. Join slots are updated but instances of the class on which the join is made are not updated.

Examples

```
(defvar *e1* (car (select 'employee :where [= [slot-value 'employee 'emplid] 1] :f
=> *E1*
(slot-value *e1* 'email)
=> "lenin@soviet.org"
(update-records [employee]
                :av-pairs '(([email] "v.lenin@soviet.org"))
                :where [= [emplid] 1])
=>
(update-instance-from-records *e1*)
=> #<EMPLOYEE {4806B53D}>
(slot-value *e1* 'email)
=> "v.lenin@soviet.org"
```

Side Effects

Slot values of *object* may be modified.

Affected by

Data in SQL database.

Exceptional Situations

If *database* is not able to be read.

See Also

```
update-slot-from-record
update-objects-joins
```

Notes

None.

Name

UPDATE-SLOT-FROM-RECORD — Update objects slot from database.
Function

Syntax

```
update-slot-from-record object slot &key database => object
```

Arguments and Values

object An instance of a *View Class*.

slot The name of a slot in *object*.

database A *database object*. This will default to the value of *default-database*.

Description

Updates the slot value, specified by the CLOS slot name *slot*, of the *View Class* instance *object* using the attribute values of the appropriate table of *database* which defaults to the database associated with *object* or, if *object* is not associated with a database, *default-database*. Join slots are updated but instances of the class on which the join is made are not updated.

Examples

```
(defvar *e1* (car (select 'employee :where [= [slot-value 'employee 'emplid] 1] :f
=> *E1*
(slot-value *e1* 'email)
=> "lenin@soviet.org"
(update-records [employee]
                :av-pairs '(([email] "v.lenin@soviet.org"))
                :where [= [emplid] 1])
=>
(update-slot-from-record *e1* 'email)
=> #<EMPLOYEE {4806B53D}>
(slot-value *e1* 'email)
=> "v.lenin@soviet.org"
```

Side Effects

Modifies the slot value of the object.

Affected by

Data in SQL database.

Exceptional Situations

Database errors.

See Also

```
update-instance-from-records
update-objects-joins
```

Notes

None.

Name

UPDATE-OBJECTS-JOINS — Updates joined slots of objects.
Function

Syntax

```
update-objects-joins objects &key slots force-p class-name max-len =>
```

Arguments and Values

objects	A list of instances of a *View Class*.
slots	* :immediate (default) - refresh join slots with :retrieval :immediate * :deferred - refresh join slots created with :retrieval :deferred * :all,t - refresh all join slots regardless of :retrieval * list of symbols - which explicit slots to refresh * a single symobl - what slot to refresh
force-p	A Boolean, defaulting to T.
class-name	A list of instances of a *View Class*.
max-len	A non-negative integer or NIL defaulting to *default-update-objects-max-len*. When non-nil this is essentially a batch size for the max number of objects to query from the database at a time. If we need more than max-len we loop till we have all the objects

Description

Updates from the records of the appropriate database tables the join slots specified by SLOTS in the supplied list of *View Class* instances OBJECTS. A simpler method of causing a join-slot to be requeried is to set it to unbound, then request it again. This function has efficiency gains where join-objects are shared among the `objects` (querying all join-objects, then attaching them appropriately to each of the `objects`)

Examples

```
(defvar *addresses* (select 'deferred-employee-address :order-by [ea_join aaddress
=> *ADDRESSES*
(slot-boundp (car *addresses*) 'address)
=> NIL
(update-objects-joins *addresses*)
=>
(slot-boundp (car *addresses*) 'address)
=> T
(slot-value (car *addresses*) 'address)
=> #<ADDRESS {480B0F1D}>
```

Side Effects

The slot values of *objects* are modified.

Affected by

default-update-objects-max-len

Exceptional Situations

Database errors.

See Also

default-update-objects-max-len
update-instance-from-records
update-slot-from-record

Notes

None.

SQL I/O Recording

CLSQL provides a facility for recording SQL commands sent to and/or results returned from the underlying RDBMS to user sprecified streams. This is useful for monitoring *CLSQL* activity and for debugging applications.

This section documents the functions provided for enabling and disabling SQL recording as well as for manipulating the streams on to which SQL commands and results are recorded.

Table of Contents

Name

START-SQL-RECORDING — Start recording SQL commands or results.
Function

Syntax

```
start-sql-recording &key type database =>
```

Arguments and Values

type One of the following keyword symbols: :commands, :results or :both, defaulting to :commands.

database A *database object*. This will default to *default-database*.

Description

Starts recording of SQL commands sent to and/or results returned from *database* which defaults to *default-database*. The SQL is output on one or more broadcast streams, initially just *standard-output*, and the functions `add-sql-stream` and `delete-sql-stream` may be used to add or delete command or result recording streams. The default value of *type* is :commands which means that SQL commands sent to *database* are recorded. If *type* is :results then SQL results returned from *database* are recorded. Both commands and results may be recorded by passing *type* value of :both.

Examples

```
(start-sql-recording :type :both)
=>
(select [last-name] :from [employee]
                    :where [= [emplid] 1]
                    :field-names nil
                    :flatp t)
;; 2004-07-02 16:42:12 dent/test/dent => SELECT last_name FROM employee WHERE (emp
;; 2004-07-02 16:42:12 dent/test/dent <= (Lenin)
=> ("Lenin")
```

Side Effects

The command and result recording broadcast streams associated with *database* are reinitialised with only *standard-output* as their component streams.

Affected by

None.

Exceptional Situations

None.

See Also

```
stop-sql-recording
sql-recording-p
sql-stream
add-sql-stream
delete-sql-stream
list-sql-streams
```

Notes

None.

Name

STOP-SQL-RECORDING — Stop recording SQL commands or results.
Function

Syntax

```
stop-sql-recording &key type database =>
```

Arguments and Values

type One of the following keyword symbols: :commands, :results or :both, defaulting to :commands.

database A *database object*. This will default to *default-database*.

Description

Stops recording of SQL commands sent to and/or results returned from *database* which defaults to *default-database*. The default value of *type* is :commands which means that SQL commands sent to *database* will no longer be recorded. If *type* is :results then SQL results returned from *database* will no longer be recorded. Recording may be stopped for both commands and results by passing *type* value of :both.

Examples

```
(start-sql-recording :type :both)
=>
(select [last-name] :from [employee]
                        :where [= [emplid] 1]
                        :field-names nil
                        :flatp t)
;; 2004-07-02 16:42:12 dent/test/dent => SELECT last_name FROM employee WHERE (emp
;; 2004-07-02 16:42:12 dent/test/dent <= (Lenin)
=> ("Lenin")
(stop-sql-recording :type :results)
=>
(select [last-name] :from [employee]
                        :where [= [emplid] 1]
                        :field-names nil
                        :flatp t)
;; 2004-07-02 16:44:11 dent/test/dent => SELECT last_name FROM employee WHERE (emp
=> ("Lenin")
```

Side Effects

The command and result recording broadcast streams associated with *database* are reinitialised to NIL.

Affected by

None.

Exceptional Situations

None.

See Also

```
start-sql-recording
sql-recording-p
```

Notes

None.

Name

SQL-RECORDING-P — Tests whether SQL commands or results are being recorded.
Function

Syntax

```
sql-recording-p &key type database => result
```

Arguments and Values

type	One of the following keyword symbols: :commands, :results, :both or :either defaulting to :commands.
database	A *database object*. This will default to *default-database*.
result	A Boolean.

Description

Predicate to test whether the SQL recording specified by *type* is currently enabled for *database* which defaults to *default-database*. *type* may be one of :commands, :results, :both or :either, defaulting to :commands, otherwise NIL is returned.

Examples

```
(start-sql-recording :type :commands)
=>
(sql-recording-p :type :commands)
=> T
(sql-recording-p :type :both)
=> NIL
(sql-recording-p :type :either)
=> T
```

Side Effects

None.

Affected by

```
start-sql-recording
stop-sql-recording
```

Exceptional Situations

None.

See Also

```
start-sql-recording
stop-sql-recording
```

Notes

The :both and :either values for the *type* keyword argument are *CLSQL* extensions.

Name

SQL-STREAM — Returns the broadcast stream used for recording SQL commands or results.
Function

Syntax

```
sql-stream &key type database => result
```

Arguments and Values

type	One of the following keyword symbols: :commands or :results, defaulting to :commands.
database	A *database object*. This will default to *default-database*.
result	A broadcast stream or NIL.

Description

Returns the broadcast stream used for recording SQL commands sent to or results returned from *database* which defaults to *default-database*. *type* must be one of :commands or :results, defaulting to :commands, and determines whether the stream returned is that used for recording SQL commands or results.

Examples

```
(start-sql-recording :type :commands)
=>
(sql-stream :type :commands)
=> #<Broadcast Stream>
(sql-stream :type :results)
=> NIL
```

Side Effects

None.

Affected by

None.

Exceptional Situations

An error is signalled if *type* is not one of :commands or :results.

See Also

start-sql-recording

```
add-sql-stream
delete-sql-stream
list-sql-streams
```

Notes

None.

Name

ADD-SQL-STREAM — Add a component to the broadcast streams used for recording SQL commands or results.
Function

Syntax

```
add-sql-stream stream &key type database => result
```

Arguments and Values

stream A stream or T.

type One of the following keyword symbols: :commands, :results or :both, defaulting to :commands.

database A *database object*. This will default to *default-database*.

result The added stream.

Description

Adds the supplied stream *stream* (or T for *standard-output*) as a component of the recording broadcast stream for the SQL recording type specified by *type* on *database* which defaults to *default-database*. *type* must be one of :commands, :results, or :both, defaulting to :commands, depending on whether the stream is to be added for recording SQL commands, results or both.

Examples

```
(start-sql-recording :type :commands)
=>
(with-output-to-string (s)
  (add-sql-stream s :type :commands)
  (print-query [select [emplid] [first-name] [last-name] [email] :from [employee]]
               :stream s))

;; 2004-07-02 17:38:45 dent/test/dent => SELECT emplid,first_name,last_name,email
=>
";; 2004-07-02 17:38:45 dent/test/dent => SELECT emplid,first_name,last_name,email
1  Vladimir   Lenin     lenin@soviet.org
2  Josef      Stalin    stalin@soviet.org
3  Leon       Trotsky   trotsky@soviet.org
4  Nikita     Kruschev  kruschev@soviet.org
5  Leonid     Brezhnev  brezhnev@soviet.org
6  Yuri       Andropov  andropov@soviet.org
7  Konstantin Chernenko chernenko@soviet.org
8  Mikhail    Gorbachev gorbachev@soviet.org
9  Boris      Yeltsin   yeltsin@soviet.org
10 Vladimir   Putin     putin@soviet.org     "
```

Side Effects

The specified broadcast stream(s) associated with *`database`* are modified.

Affected by

None.

Exceptional Situations

None.

See Also

```
start-sql-recording
sql-stream
delete-sql-stream
list-sql-streams
```

Notes

None.

Name

DELETE-SQL-STREAM — Remove a component from the broadcast streams used for recording SQL commands or results.
Function

Syntax

```
delete-sql-stream stream &KEY type database => result
```

Arguments and Values

stream	A stream or T.
stream	A stream or T.
type	One of the following keyword symbols: :commands, :results or :both, defaulting to :commands.
database	A *database object*. This will default to *default-database*.
result	The added stream.

Description

Removes the supplied stream *stream* from the recording broadcast stream for the SQL recording type specified by *type* on *database* which defaults to *default-database*. *type* must be one of :commands, :results, or :both, defaulting to :commands, depending on whether the stream is to be added for recording SQL commands, results or both.

Examples

```
(list-sql-streams :type :both)
=> (#<Stream for descriptor 7> #<Stream for descriptor 7>)
(delete-sql-stream *standard-output* :type :results)
=> #<Stream for descriptor 7>
(list-sql-streams :type :both)
=> (#<Stream for descriptor 7>)
```

Side Effects

The specified broadcast stream(s) associated with *database* are modified.

Affected by

None.

Exceptional Situations

None.

See Also

```
start-sql-recording
stop-sql-recording
sql-recording-p
sql-stream
add-sql-stream
delete-sql-stream
list-sql-streams
```

Notes

None.

Name

LIST-SQL-STREAMS — List the components of the broadcast streams used for recording SQL commands or results.
Function

Syntax

```
list-sql-streams &key type database => result
```

Arguments and Values

type	One of the following keyword symbols: :commands, :results or :both, defaulting to :commands.
database	A *database object*. This will default to *default-database*.
result	A list.

Description

Returns the list of component streams for the broadcast stream recording SQL commands sent to and/or results returned from *database* which defaults to *default-database*. *type* must be one of :commands, :results, or :both, defaulting to :commands, and determines whether the listed streams contain those recording SQL commands, results or both.

Examples

```
(list-sql-streams :type :both)
=> NIL
(start-sql-recording :type :both)
=>
(list-sql-streams :type :both)
=> (#<Stream for descriptor 7> #<Stream for descriptor 7>)
```

Side Effects

None.

Affected by

```
add-sql-stream
delete-sql-stream
```

Exceptional Situations

An error is signalled if *type* is passed a value other than :commands, :results or :both.

See Also

```
sql-stream
add-sql-stream
delete-sql-stream
```

Notes

None.

CLSQL Condition System

CLSQL provides and uses a condition system in which all errors and warnings are of type sql-condition. This section describes the various subclasses of sql-condition defined by *CLSQL*. Details are also provided for how they are used in *CLSQL* and intended to be signalled in user code. Finally, slot accessors for some of the condition types are described.

Table of Contents

Name

BACKEND-WARNING-BEHAVIOR — Controls behaviour on warnings from underlying RDBMS.
Variable

Value Type

Meaningful values are :warn, :error, :ignore and `NIL`.

Initial Value

:warn

Description

Action to perform on warning messages from backend. Default is to :warn. May also be set to :error to signal an error or :ignore or `NIL` to silently ignore the warning.

Examples

Affected By

None.

See Also

None.

Notes

backend-warning-behaviour is a *CLSQL* extension.

Name

SQL-CONDITION — the super-type of all *CLSQL*-specific conditions
Condition Type

Class Precedence List

sql-condition, condition, t

Description

This is the super-type of all *CLSQL*-specific conditions defined by *CLSQL*, or any of it's database-specific interfaces. There are no defined initialization arguments nor any accessors.

Notes

sql-condition is a *CLSQL* extension.

Name

SQL-ERROR — the super-type of all *CLSQL*-specific errors
Condition Type

Class Precedence List

sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This is the super-type of all *CLSQL*-specific conditions that represent errors, as defined by *CLSQL*, or any of it's database-specific interfaces. There are no defined initialization arguments nor any accessors.

Notes

sql-error is a *CLSQL* extension.

Name

SQL-WARNING — the super-type of all *CLSQL*-specific warnings
Condition Type

Class Precedence List

sql-warning, warning, sql-condition, condition, t

Description

This is the super-type of all *CLSQL*-specific conditions that represent warnings, as defined by *CLSQL*, or any of it's database-specific interfaces. There are no defined initialization arguments nor any accessors.

Notes

sql-warning is a *CLSQL* extension.

Name

SQL-DATABASE-WARNING — Used to warn while accessing a *CLSQL* database.
Condition Type

Class Precedence List

sql-database-warning, sql-warning, warning, sql-condition, condition, t

Description

This condition represents warnings signalled while accessing a database.

The following initialization arguments and accessors exist:
Initarg: :database
Accessor: `sql-warning-database`
Description: The database object that was involved in the incident.

Notes

sql-database-warning is a *CLSQL* extension.

Name

SQL-USER-ERROR — condition representing errors because of invalid parameters from the library user.
Condition Type

Class Precedence List

sql-user-error, sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors that occur because the user supplies invalid data to *CLSQL*. This includes errors such as an invalid format connection specification or an error in the syntax for the `LOOP` macro extensions.

The following initialization arguments and accessors exist:
Initarg: :message
Accessor: `sql-user-error-message`
Description: The error message.

Notes

The slot accessor `sql-user-error-message` is a *CLSQL* extension.

Name

SQL-DATABASE-ERROR — condition representing errors during query or command execution
Condition Type

Class Precedence List

sql-database-error, sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors that occur while executing SQL statements, either as part of query operations or command execution, either explicitly or implicitly, as caused e.g. by `with-transaction`.

The following initialization arguments and accessors exist:
Initarg: :database
Accessor: `sql-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Notes

The slot accessor `sql-error-database` is a *CLSQL* extension.

Name

SQL-CONNECTION-ERROR — condition representing errors during connection
Condition Type

Class Precedence List

sql-connection-error, sql-database-error, sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors that occur while trying to connect to a database.

The following initialization arguments and accessors exist:
Initarg: :database-type
Accessor: `sql-error-database-type`
Description: Database type for the connection attempt
Initarg: :connection-spec
Accessor: `sql-error-connection-spec`
Description: The connection specification used in the connection attempt.
Initarg: :database
Accessor: `sql-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-database-error-error`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Notes

The slot accessors `sql-error-database`, `sql-error-database-type` and `sql-error-connection-spec` are *CLSQL* extensions.

Name

SQL-DATABASE-DATA-ERROR — Used to signal an error with the SQL data passed to a database.
Condition Type

Class Precedence List

sql-database-data-error, sql-database-error, sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors that occur while executing SQL statements, specifically as a result of malformed SQL expressions.

The following initialization arguments and accessors exist:
Initarg: :expression
Accessor: `sql-error-expression`
Description: The SQL expression whose execution caused the error.
Initarg: :database
Accessor: `sql-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Notes

The slot accessors `sql-error-database` and `sql-error-expression` are *CLSQL* extensions.

Name

SQL-TEMPORARY-ERROR — Used to signal a temporary error in the database backend.
Condition Type

Class Precedence List

sql-temporary-error, sql-database-error, sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors occurring when the database cannot currently process a valid interaction because, for example, it is still executing another command possibly issued by another user.

The following initialization arguments and accessors exist:
Initarg: :database
Accessor: `sql-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Notes

The slot accessor `sql-error-database` is a *CLSQL* extension.

Name

SQL-TIMEOUT-ERROR — condition representing errors when a connection times out.
Condition Type

Class Precedence List

sql-connection-error, sql-database-error, sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors that occur when the database times out while processing some operation. The following initialization arguments and accessors exist:
Initarg: :database-type
Accessor: `sql-error-database-type`
Description: Database type for the connection attempt
Initarg: :connection-spec
Accessor: `sql-error-connection-spec`
Description: The connection specification used in the connection attempt.
Initarg: :database
Accessor: `sql-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Notes

The slot accessors `sql-error-database`, `sql-error-database-type` and `sql-error-connection-spec` are *CLSQL* extensions.

Name

SQL-FATAL-ERROR — condition representing a fatal error in a database connection
Condition Type

Class Precedence List

sql-connection-error, sql-database-error, sql-error, simple-error, simple-condition, error, serious-condition, sql-condition, condition, t

Description

This condition represents errors occurring when the database connection is no longer usable.

The following initialization arguments and accessors exist:
Initarg: :database-type
Accessor: `sql-error-database-type`
Description: Database type for the connection attempt
Initarg: :connection-spec
Accessor: `sql-error-connection-spec`
Description: The connection specification used in the connection attempt.
Initarg: :database
Accessor: `sql-error-database`
Description: The database object that was involved in the incident.
Initarg: :error-id
Accessor: `sql-error-error-id`
Description: The numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :secondary-error-id
Accessor: `sql-error-secondary-error-id`
Description: The secondary numeric or symbolic error specification returned by the database back-end. The values and semantics of this are interface specific.
Initarg: :message
Accessor: `sql-error-database-message`
Description: A string describing the problem that occurred, possibly one returned by the database back-end.

Notes

The slot accessors `sql-error-database`, `sql-error-database-type` and `sql-error-connection-spec` are *CLSQL* extensions.

Index

Table of Contents

Name

Alphabetical Index for package CLSQL — Clickable index of all symbols

Appendix A. Database Back-ends

How CLSQL finds and loads foreign libraries

For some database types CLSQL has to load external foreign libaries. These are usually searched for in the standard locations the operating system uses but you can tell *CLSQL* to look into other directories as well by using the function `CLSQL:PUSH-LIBRARY-PATH` or by directly manipulating the special variable `CLSQL:*FOREIGN-LIBRARY-SEARCH-PATHS*`. If, say, the shared library libpq.so needed for PostgreSQL support is located in the directory `/opt/foo/` on your machine you'd use

```
(clsql:push-library-path "/opt/foo/")
```

before loading the CLSQL-POSTGRESQL module. (Note the trailing slash above!) If you want to combine this with fully automatic loading of libraries via ASDF a technique like the following works:

```
(defmethod asdf:perform :after ((o asdf:load-op)
                                (c (eql (asdf:find-system 'clsql))))
  (funcall (find-symbol (symbol-name '#:push-library-path)
                        (find-package 'clsql))
           #p"/opt/foo/"))
```

Additionally, site-specific initialization can be done using an initialization file. If the file `/etc/clsql-init.lisp` exists, this file will be read after the *CLSQL* ASDF system is loaded. This file can contain forms to set site-specific paths as well as change *CLSQL* default values.

PostgreSQL

Libraries

The PostgreSQL back-end requires the PostgreSQL C client library (`libpq.so`). The location of this library is specified via *postgresql-so-load-path*, which defaults to `/usr/lib/libpq.so`. Additional flags to ld needed for linking are specified via *postgresql-so-libraries*, which defaults to ("-lcrypt" "-lc").

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-postgresql)
```

to load the PostgreSQL back-end. The database type for the PostgreSQL back-end is :postgresql.

Connection Specification

Syntax of connection-spec

```
(host db user password &optional port options tty)
```

Description of connection-spec

For every parameter in the connection-spec, nil indicates that the PostgreSQL default environment variables (see PostgreSQL documentation) will be used, or if those are unset, the compiled-in defaults of the C client library are used.

host String representing the hostname or IP address the PostgreSQL server resides on. Use the empty string to indicate a connection to localhost via Unix-Domain sockets instead of TCP/IP.

db String representing the name of the database on the server to connect to.

user String representing the user name to use for authentication.

password String representing the unencrypted password to use for authentication.

port String representing the port to use for communication with the PostgreSQL server.

options String representing further runtime options for the PostgreSQL server.

tty String representing the tty or file to use for debugging messages from the PostgreSQL server.

Notes

None.

PostgreSQL Socket

Libraries

The PostgreSQL Socket back-end needs *no* access to the PostgreSQL C client library, since it communicates directly with the PostgreSQL server using the published frontend/backend protocol, version 2.0. This eases installation and makes it possible to dump CMU CL images containing CLSQL and this backend, contrary to backends which require FFI code.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-postgresql-socket)
```

to load the PostgreSQL Socket back-end. The database type for the PostgreSQL Socket back-end is :postgresql-socket.

Connection Specification

Syntax of connection-spec

```
(host db user password &optional port options tty)
```

Description of connection-spec

host If this is a string, it represents the hostname or IP address the PostgreSQL server resides on. In this case communication with the server proceeds via a TCP connection to the given host and port.

If this is a pathname, then it is assumed to name the directory that contains the server's Unix-Domain sockets. The full name to the socket is then constructed from this and the port number passed, and communication will proceed via a connection to this unix-domain socket.

db String representing the name of the database on the server to connect to.

user String representing the user name to use for authentication.

password String representing the unencrypted password to use for authentication. This can be the empty string if no password is required for authentication.

port Integer representing the port to use for communication with the PostgreSQL server. This defaults to 5432.

options String representing further runtime options for the PostgreSQL server.

tty String representing the tty or file to use for debugging messages from the PostgreSQL server.

Notes

None.

MySQL

Libraries

The MySQL back-end requires the MySQL C client library (`libmysqlclient.so`). The location of this library is specified via *mysql-so-load-path*, which defaults to `/usr/lib/libmysqlclient.so`. Additional flags to ld needed for linking are specified via *mysql-so-libraries*, which defaults to ("-lc").

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-mysql)
```

to load the MySQL back-end. The database type for the MySQL back-end is :mysql.

Connection Specification

Syntax of connection-spec

(*host db user password* &optional *port*)

Description of connection-spec

host String representing the hostname or IP address the MySQL server resides on, or nil to indicate the localhost.

db String representing the name of the database on the server to connect to.

user String representing the user name to use for authentication, or nil to use the current Unix user ID.

password String representing the unencrypted password to use for authentication, or nil if the authentication record has an empty password field.

port String representing the port to use for communication with the MySQL server.

Notes

FDDL

- `drop-index` requires a table to be specified with the :on keyword parameter.
- *views* are not supported by MySQL.
- The :transactions keyword argument to `create-table` controls whether or not the created table is an InnoDB table which supports transactions.
- The :owner keyword argument to the FDDL functions for listing and testing for database objects is ignored.

FDML

- Prior to version 4.1, MySQL does not support nested subqueries in calls to `select`.

Symbolic SQL Syntax

- MySQL does not support the `||` concatenation operator. Use `concat` instead.
- MySQL does not support the `substr` operator. Use `substring` instead.
- MySQL does not support the `intersect` and `except` set operations.
- MySQL (version 4.0 and later) does not support string table aliases unless the server is started with ANSI_QUOTES enabled.

ODBC

Libraries

The ODBC back-end requires access to an ODBC driver manager as well as ODBC drivers for the underlying database server. *CLSQL* has been tested with unixODBC ODBC Driver Manager as well as Microsoft's ODBC manager. These driver managers have been tested with the *psqlODBC* [http://odbc.postgresql.org] driver for PostgreSQL and the *MyODBC* [http://www.mysql.com/products/connector/odbc/] driver for MySQL.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-odbc)
```

to load the ODBC back-end. The database type for the ODBC back-end is :odbc.

Connection Specification

Syntax of connection-spec

(*dsn user password* &key *connection-string*)

Description of connection-spec

dsn	String representing the ODBC data source name.
user	String representing the user name to use for authentication.
password	String representing the unencrypted password to use for authentication.
connection-string	Raw connection string passed to the underlying ODBC driver. Allows bypassing creating a DSN on the server.

Notes

FDDL

- The :owner keyword argument to the FDDL functions for listing and testing for database objects is ignored.

Connect Examples

```
;; assumes a "mssql" DSN is configured on the lisp host, specifying database serve
;;  and database name.
> (clsql:connect '("mssql" "database-user" "database-password")
                 :database-type :odbc)
=> #<CLSQL-ODBC:ODBC-DATABASE mssql/database-user OPEN {100756D123}>

;; no DSN on the lisp host, specify connection information via :connection-string
> (clsql:connect '("friendly-server-name" "friendly-username" ""
   :connection-string "DRIVER={FreeTDS};SERVER=mssql-server;DATABASE=database-name
                 :database-type :odbc)
=> #<CLSQL-ODBC:ODBC-DATABASE friendly-server-name/friendly-username OPEN {100756D
```

The friendly-server-name and friendly-username are only used when printing the connection object to a stream.

AODBC

Libraries

The AODBC back-end requires access to the ODBC interface of AllegroCL named DBI. This interface is not available in the trial version of AllegroCL

Initialization

Use

```
(require 'aodbc-v2)
(asdf:operate 'asdf:load-op 'clsql-aodbc)
```

to load the AODBC back-end. The database type for the AODBC back-end is :aodbc.

Connection Specification

Syntax of connection-spec

(*dsn user password*)

Description of connection-spec

dsn String representing the ODBC data source name.

user String representing the user name to use for authentication.

password String representing the unencrypted password to use for authentication.

Notes

None.

SQLite version 2

Libraries

The SQLite version 2 back-end requires the SQLite version 2 shared library file. Its default file name is `/usr/lib/libsqlite.so`.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-sqlite)
```

to load the SQLite version 2 back-end. The database type for the SQLite version 2 back-end is :sqlite.

Connection Specification

Syntax of connection-spec

(*filename*)

Description of connection-spec

filename String or pathname representing the filename of the SQLite version 2 database file.

Notes

Connection

- Passing *filename* a value of `:memory:` will create a database in physical memory instead of using a file on disk.
- Some operations will be many times faster if database integrity checking is disabled by setting the SYNCHRONOUS flag to OFF (see the SQLITE manual for details).

FDDL

- The :owner keyword argument to the FDDL functions for listing and testing for database objects is ignored.
- The :column-list keyword argument to `create-view` is not supported by SQLite version 2.

Symbolic SQL Syntax

- SQLite version 2 does not support the `all`, `some`, `any` and `exists` subquery operations.

SQLite version 3

Libraries

The SQLite version 3 back-end requires the SQLite version 3 shared library file. Its default file name is `/usr/lib/libsqlite3.so`.

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-sqlite3)
```

to load the SQLite version 3 back-end. The database type for the SQLite version 3 back-end is :sqlite3.

Connection Specification

Syntax of connection-spec

(*filename* &optional *init-function*)

Description of connection-spec

filename String representing the filename of the SQLite version 3 database file.

init-function A function designator. *init-function* takes a single argument of type sqlite3:sqlite3-db, a foreign pointer to the C descriptor of the newly opened database. *init-function* is called by the back-end immediately after SQLite version 3 `sqlite3_open` library function, and can be used to perform optional database initializations by calling foreign functions in the SQLite version 3 library.

An example of an initialization function which defines a new collating sequence for text columns is provided in `./examples/sqlite3/init-func/`.

Notes

Connection

- Passing *filename* a value of `:memory:` will create a database in physical memory instead of using a file on disk.
- Some operations will be many times faster if database integrity checking is disabled by setting the SYNCHRONOUS flag to OFF (see the SQLITE manual for details).

FDDL

- The :owner keyword argument to the FDDL functions for listing and testing for database objects is ignored.
- The :column-list keyword argument to `create-view` is not supported by SQLite version 3.

Symbolic SQL Syntax

- SQLite version 3 does not support the `all`, `some`, `any` and `exists` subquery operations.

Oracle

Libraries

The Oracle back-end requires the Oracle OCI client library. (`libclntsh.so`). The location of this library is specified relative to the ORACLE_HOME value in the operating system environment.

Library Versions

CLSQL has tested sucessfully using the client library from Oracle 9i and Oracle 10g server installations as well as Oracle's 10g Instant Client library. For Oracle 8 and earlier versions, there is vestigial support by pushing the symbol :oci7 onto cl:*features* prior to loading the `clsql-oracle` ASDF system.

```
(push :oci7 cl:*features*)
(asdf:operate 'asdf:load-op 'clsql-oracle)
```

Initialization

Use

```
(asdf:operate 'asdf:load-op 'clsql-oracle)
```

to load the Oracle back-end. The database type for the Oracle back-end is :oracle.

Connection Specification

Syntax of connection-spec

(*global-name user password*)

Description of connection-spec

global-name	String representing the global name of the Oracle database. This is looked up through the tnsnames.ora file.
user	String representing the user name to use for authentication.
password	String representing the password to use for authentication..

Notes

Symbolic SQL Syntax

- The `userenv` operator is Oracle specific.
- Oracle does not support the `except` operator. Use `minus` instead.
- Oracle does not support the `all`, `some`, `any` subquery operations.

Transactions

- By default, *CLSQL* starts in transaction AUTOCOMMIT mode (see `set-autocommit`). To begin a transaction in autocommit mode, `start-transaction` has to be called explicitly.

Glossary

Note

This glossary is still very thinly populated, and not all references in the main text have been properly linked and coordinated with this glossary. This will hopefully change in future revisions.

Attribute	A property of objects stored in a database table. Attributes are represented as columns (or fields) in a table.
Active database	See Database Object.
Connection	See Database Object.
Column	See Attribute.
Data Definition Language (DDL)	The subset of SQL used for defining and examining the structure of a database.
Data Manipulation Language (DML)	The subset of SQL used for inserting, deleting, updating and fetching data in a database.
database	See Database Object.
Database Object	An object of type database.
Field	See Attribute.
Field Types Specifier	A value that specifies the type of each field in a query.
Foreign Function Interface (FFI)	An interface from Common Lisp to a external library which contains compiled functions written in other programming languages, typically C.
Query	An SQL statement which returns a set of results.
RDBMS	A Relational DataBase Management System (RDBMS) is a software package for managing a database in which the data is defined, organised and accessed as rows and columns of a table.
Record	A sequence of attribute values stored in a database table.
Row	See Record.
Structured Query Language (SQL)	An ANSI standard language for storing and retrieving data in a relational database.
SQL Expression	Either a string containing a valid SQL statement, or an object of type sql-expression.
Table	A collection of data which is defined, stored and accessed as tuples of attribute values (i.e., rows and columns).
Transaction	An atomic unit of one or more SQL statements of which all or none are successfully executed.
Tuple	See Record.

View	A table display whose structure and content are derived from an existing table via a query.
View Class	The class `standard-db-object` or one of its subclasses.