DB: Database Connectivity

Version 6.2

Ryan Culpepper <ryanc@racket-lang.org>

June 19, 2015

A database interface for functional programmers.

```
(require db)        package: db-lib
```

This library provides a high-level interface to several database systems. The following database systems are currently supported:

- **PostgreSQL versions 7.4 and later.** This library implements the PostgreSQL wire protocol, so no native client library is required.
- **MySQL versions 5 and later.** This library implements the MySQL wire protocol, so no native client library is required.
- **SQLite version 3.** The SQLite native client library is required; see §6.6 "SQLite Requirements".
- **ODBC.** An ODBC Driver Manager and appropriate ODBC drivers are required; see §6.8 "ODBC Requirements". The following database systems are known to work with this library via ODBC (see §6.9 "ODBC Status" for details): **DB2**, **Oracle**, and **SQL Server**.

The query operations are functional in spirit: queries return results or raise exceptions rather than stashing their state into a cursor object for later navigation and retrieval. Query parameters and result fields are automatically translated to and from appropriate Racket values. Resources are managed automatically by the garbage collector and via custodians. Connections are internally synchronized, so multiple threads can use a connection simultaneously.

Acknowledgments Thanks to Dave Gurnell, Noel Welsh, Mike Burns, and Doug Orleans for contributions to `spgsql`, the PostgreSQL-only predecessor of this library. The SQLite support is based in part on code from Jay McCarthy's `sqlite` package.

1 Using Database Connections

This section introduces this library's basic features and covers some practical issues with database programming in general and with this library in particular.

1.1 Introduction to Using Database Connections

The following annotated program demonstrates how to connect to a database and perform simple queries. Some of the SQL syntax used below is PostgreSQL-specific, such as the syntax of query parameters (`$1` rather than `?`).

```
> (require db)
```

First we create a connection. Replace *user*, *db*, and *password* below with the appropriate values for your configuration (see §2.1 "Base Connections" for other connection examples):

```
> (define pgc
    (postgresql-connect #:user user
                        #:database db
                        #:password password))
```

Use `query-exec` method to execute a SQL statement for effect.

```
> (query-exec pgc
   "create temporary table the_numbers (n integer, d var-
char(20))")

> (query-exec pgc
    "insert into the_numbers values (0, 'nothing')")

> (query-exec pgc
    "insert into the_numbers values (1, 'the loneliest number')")
```

When a query contains a SQL value that isn't constant, refer to it through a "query parameter" rather than by dynamically computing the SQL query string (see also §1.2.1 "SQL Injection"). Just provide the parameter values after the SQL statement in the query function call:

```
> (query-exec pgc
    "insert into the_numbers values ($1, $2)"
```

```
    (+ 1 1)
    "company")
```

Every standard query function accepts query parameters. The SQL syntax for query parameters depends on the database system (see §3.1 “Statements”). Other options for running parameterized queries are discussed below.

The `query` function is a more general way to execute a statement. It returns a structure encapsulating information about the statement’s execution. (But some of that information varies from system to system and is subject to change.)

```
> (query pgc "insert into the_numbers values (3, 'a crowd')")
(simple-result '((insert-id . #f) (affected-rows . 1)))
> (query pgc "select n, d from the_numbers where n % 2 = 0")
(rows-result
 '(((name . "n") (typeid . 23) (type-size . 4) (type-mod . -1))
   ((name . "d") (typeid . 1043) (type-size . -1) (type-mod .
24)))
 '(#(0 "nothing") #(2 "company")))
```

When the query is known to return rows and when the field descriptions are not needed, it is more convenient to use the `query-rows` function.

```
> (query-rows pgc "select n, d from the_numbers where n % 2 = 0")
'(#(0 "nothing") #(2 "company"))
```

Use `query-row` for queries that are known to return exactly one row.

```
> (query-row pgc "select * from the_numbers where n = 0")
'#(0 "nothing")
```

Similarly, use `query-list` for queries that produce rows of exactly one column.

```
> (query-list pgc "select d from the_numbers order by n")
'("nothing" "the loneliest number" "company" "a crowd")
```

When a query is known to return a single value (one row and one column), use `query-value`.

```
> (query-value pgc "select count(*) from the_numbers")
4
> (query-value pgc "select d from the_numbers where n = 5")
query-value: query returned wrong number of rows
  statement: "select d from the_numbers where n = 5"
  expected: 1
  got: 0
```

When a query may return zero or one rows, as the last example, use query-maybe-row or query-maybe-value instead.

```
> (query-maybe-value pgc "select d from the_numbers where n = 5")
#f
```

The in-query function produces a sequence that can be used with Racket's iteration forms:

```
> (for ([(n d) (in-query pgc "select * from the_numbers where n <
4")])
    (printf "~a: ~a\n" n d))
0: nothing
1: the loneliest number
2: company
3: a crowd

> (for/fold ([sum 0]) ([n (in-query pgc "select n from
the_numbers")])
    (+ sum n))
6
```

Errors in queries generally do not cause the connection to disconnect.

```
> (begin (with-handlers [(exn:fail?
                          (lambda (e)
                            (printf "~a~n" (exn-message e))))]
           (query-value pgc "select NoSuchField from No-
SuchTable"))
         (query-value pgc "select 'okay to proceed!'"))
query-value: relation "nosuchtable" does not exist
  SQLSTATE: 42P01
"okay to proceed!"
```

Queries may contain parameters. The easiest way to execute a parameterized query is to provide the parameters "inline" after the SQL statement in the query function call.

```
> (query-value pgc
   "select d from the_numbers where n = $1" 2)
"company"
> (query-list pgc
   "select n from the_numbers where n > $1 and n < $2" 0 3)
'(1 2)
```

Alternatively, a parameterized query may be prepared in advance and executed later. Prepared statements can be executed multiple times with different parameter values.

```
> (define get-less-than-pst
    (prepare pgc "select n from the_numbers where n < $1"))

> (query-list pgc get-less-than-pst 1)
'(0)
> (query-list pgc (bind-prepared-statement get-less-than-
pst '(2)))
'(0 1)
```

When a connection's work is done, it should be disconnected.

```
> (disconnect pgc)
```

1.2 Database Security

Database security requires both that the database back end be secured against unauthorized use and that authorized clients are not tricked or subverted into violating the database's security.

Securing database back ends is mostly beyond the scope of this manual. In brief: choose sufficiently strong authentication methods and keep credentials secure, and follow the principle of least privilege: create and use roles that have the minimum permissions needed.

The following is an incomplete list of security issues related to database *client* programming.

1.2.1 SQL Injection

SQL injection happens when part of a SQL statement that was intended as SQL literal data is instead interpreted as SQL code—possibly malicious SQL code.

Avoid dynamically creating SQL query strings by string concatenation or interpolation (eg, with `string-append` or `format`). In most cases, it is possible to use parameterized queries instead. For example, instead of this

```
; WRONG! DANGER!
(query-exec c
  (format "UPDATE users SET passwd='~a' WHERE user='~a'"
          user new-passwd))
```

write one of the following instead (depending on SQL dialect):

```
; for PostgreSQL, SQLite
(query-exec c "UPDATE users SET passwd=$1 WHERE user=$2" user new-
passwd)
; for MySQL, SQLite, ODBC
(query-exec c "UPDATE users SET passwd=? WHERE user=?" user new-
passwd)
```

The first form would choke on names like `"Patrick O'Connor"`. Worse, it would be susceptible to attack by malicious input like `"me' OR user='root'"`, which yields the following SQL statement:

```
UPDATE users SET passwd='whatever' WHERE user='me' OR user='root'
```

In contrast, using a parameterized query causes the parameterized SQL and its arguments to be submitted to the back end separately; the back end then combines them safely.

Only SQL literal values can be replaced with parameter placeholders; a SQL statement cannot be parameterized over a column name or a sort order, for example. In such cases, constructing the query dynamically may be the only feasible solution. But while the query construction may be influenced by external input, it should never directly incorporate external input without validation. That is, don't do the following:

```
; WRONG! DANGER!
(query-rows c
  (format "SELECT name, ~a FROM contestants" column))
(query-list c
  (format "SELECT name FROM contestants ORDER BY score
~a" direction))
```

Instead, select the inserted SQL from known good alternatives:

```
; BETTER
(query-rows c
  (format "SELECT name, ~a FROM contestants"
          (cond [(member column '("wins" "losses")) column]
                [else (error ....)])))
(query-list c
  (format "SELECT name FROM contestants ORDER BY score ~a"
          (if ascending? "ASC" "DESC")))
```

1.2.2 Cross-site Scripting (XSS)

Cross-site scripting—which should probably be called "HTML injection" or "markup injection"—is when arbitrary text from an untrusted source is embedded without escaping into an HTML page. The *unstructured text from the untrusted source* is reinterpreted as *markup from the web server*; if the reinterpreted markup contains embedded Javascript code, it executes with the security privileges associated with the web server's domain.

This issue has little to do with databases *per se* except that such text is often stored in a database. This issue is mitigated by using structured markup representations like SXML or X-expressions (xexprs), since they automatically escape "markup" characters found in embedded text.

1.3 Database Performance

Achieving good database performance mostly consists of good database design and intelligent client behavior.

On the database design side, most important are wise use of indexes and choosing appropriate data representations. As an example of the latter, a regexp-based search using LIKE will probably be slower than a specialized full-text search feature for large data sets. Consult your database back end's manual for additional performance advice.

The following sections describe a few client-side aspects of performance.

1.3.1 The N+1 Selects Problem

A common mistake is to fetch a large amount of data by running a query to get a set of initial records and then running another query inside a loop with an iteration for each of the initial records. This is sometimes called the "n+1 selects problem." For example:

```
(for/list ([(name id) (in-query c "SELECT name, id FROM contes-
tants")])
  (define wins
    (query-list c "SELECT contest FROM contests WHERE winner =
$1" id))
  (make-contestant-record name wins))
```

The same information can be retrieved in a single query by performing a LEFT OUTER JOIN and grouping the results:

```
(for/list ([(name id wins)
            (in-query c
             (string-append "SELECT name, id, contest "
                            "FROM contestants LEFT OUTER JOIN con-
tests "
                            "ON contestants.id = contests.winner")
             #:group '(#("name" "id"))
```

```
              #:group-mode '(list))])
   (make-contestant-record name wins))
```

The one-query form will perform better when database communication has high latency. On the other hand, it may duplicate the contents of the non-key `name` column, using more bandwidth. Another approach is to perform two queries:

```
(let ([id=>name
       (rows->dict #:key "id" #:value "name"
        (query c "SELECT id, name FROM contestants"))])
  (for/list ([(id wins)
              (in-query c
               (string-append "SELECT id, contest "
                              "FROM contestants LEFT OUTER JOIN
contests "
                              "ON contestants.id =
contests.winner")
              #:group '(#("id"))
              #:group-mode '(list))])
    (make-contestant-record (dict-ref id=>name id) wins)))
```

Compared with the one-query form, the two-query form requires additional communication, but it avoids duplicating `name` values in the `OUTER JOIN` results. If additional non-key `contestant` fields were to be retrieved, the bandwidth savings of this approach would be even greater.

See also §1.3.4 "Testing Performance of Database-Backed Programs".

1.3.2 Updates and Transactions

Using transactions can dramatically improve the performance of bulk database operations, especially `UPDATE` and `INSERT` statements. As an extreme example, on commodity hardware in 2012, SQLite is capable of executing thousands of `INSERT` statements per second within a transaction, but it is capable of only dozens of single-`INSERT` transactions per second.

1.3.3 Statement Caching

Connections cache implicitly prepared statements (that is, statements given in string form directly to a query function). The effect of the cache is to eliminate an extra round-trip to the server (to send the statement and receive a prepared statement handle), leaving just a single round-trip (to send parameters and receive results) per execution.

Currently, prepared statements are only cached within a transaction. The statement cache is flushed when entering or leaving a transaction and whenever a DDL statement is executed.

1.3.4 Testing Performance of Database-Backed Programs

When testing the performance of database-backed programs, remember to test them in environments with realistic latency and bandwidth. High-latency environments may be roughly approximated with the `high-latency-connection` function, but there's no substitute for the real thing.

1.3.5 Transactions and Concurrency

Database systems use transactions to guarantee properties such as atomicity and isolation while accommodating concurrent reads and writes by the database's clients. Within a transaction a client is insulated from the actions of other clients, but the transaction may be aborted and rolled back if the database system cannot reconcile it with other concurrent interactions. Some database systems are more adept at reconciling transactions than others, and most allow reconciliation to be tuned through the specification of *isolation levels*.

PostgreSQL supports very fine-grained reconciliation: two transactions that both read and modify the same table concurrently might both be allowed to complete if they involve disjoint sets of rows. However, clients should be prepared to retry transactions that fail with a `exn:fail:sql` exception with SQLSTATE matching `#rx"^40...$"`—typically `"40001"`, "could not serialize access due to concurrent update."

MySQL's transaction behavior varies based on the storage drivers in use. Clients should be prepared to retry transactions that fail with a `exn:fail:sql` exception with SQLSTATE matching `#rx"^40...$"`.

SQLite enforces a very coarse-grained policy: only one transaction is allowed to write to the database at a time, and thus concurrent writers are very likely to conflict. Clients should be prepared to retry transactions that fail with a `exn:fail:sql` exception with SQLSTATE of `'busy`.

An alternative to retrying whole SQLite transactions is to start each transaction with the appropriate locking level, since a transaction usually fails when it is unable to upgrade its lock level. Start a transaction that only performs reads in the default mode, and start a transaction that may perform writes in `'immediate` mode (see `start-transaction`). That converts the problem of retrying whole transactions into the problem of retrying the initial `BEGIN TRANSACTION` statment, and this library already automatically retries individual statements that fail with `'busy` errors. Depending on the length and frequency of the transactions, you may need to adjust *`busy-retry-limit`* (see `sqlite3-connect`).

ODBC's behavior varies depending on the driver and back end. See the appropriate database system's documentation.

1.4 Databases and Web Servlets

Using database connections in a web servlet is more complicated than in a standalone program. A single servlet potentially serves many requests at once, each in a separate request-handling thread. Furthermore, the use of send/suspend, send/suspend/dispatch, etc means that there are many places where a servlet may start and stop executing to service a request.

Why not use a single connection to handle all of a servlet's requests? That is, create the connection with the servlet instance and never disconnect it. Such a servlet would look something like the following:

"bad-servlet.rkt"

```
#lang web-server
(require db)
(define db-conn (postgresql-connect ....))
(define (serve req)
  .... db-conn ....)
```

The main problem with using one connection for all requests is that multiple threads accessing the same connection are not properly isolated. For example, if one thread is accessing the connection within a transaction and another thread issues a query, the second thread may see invalid data or even disrupt the work of the first thread.

A secondary problem is performance. A connection can only perform a single query at a time, whereas most database systems are capable of concurrent query processing.

The proper way to use database connections in a servlet is to create a connection for each request and disconnect it when the request has been handled. But since a request thread may start and stop executing in many places (due to send/suspend, etc), inserting the code to connect and disconnect at the proper places can be challenging and messy.

A better solution is to use a virtual connection, which automatically creates a request-specific (that is, thread-specific) "actual connection" by need and disconnects it when the request has been handled (that is, when the thread terminates). Different request-handling threads using the same virtual connection are assigned different actual connections, so the requests are properly isolated.

"better-servlet.rkt"

```
#lang web-server
(require db)
(define db-conn
  (virtual-connection
```

```
    (lambda () (postgresql-connect ....))))
(define (serve req)
  .... db-conn ....)
```

This solution preserves the simplicity of the naive solution and fixes the isolation problem but at the cost of creating many short-lived database connections. That cost can be eliminated by using a connection pool:

"best-servlet.rkt"

```
#lang web-server
(require db)
(define db-conn
  (virtual-connection
   (connection-pool
    (lambda () (postgresql-connect ....)))))
(define (serve req)
  .... db-conn ....)
```

By using a virtual connection backed by a connection pool, a servlet can achieve simplicity, isolation, and performance all at the same time.

2 Connections

This section describes functions for creating connections as well as administrative functions for managing connections.

2.1 Base Connections

There are four kinds of base connection, and they are divided into two groups: *wire-based connections* and *FFI-based connections*. PostgreSQL and MySQL connections are wire-based, and SQLite and ODBC connections are FFI-based. See also §6.7 “FFI-Based Connections and Concurrency”.

Base connections are made using the following functions.

```
(postgresql-connect
  #:user user
  #:database database
 [#:server server
  #:port port
  #:socket socket
  #:password password
  #:allow-cleartext-password? allow-cleartext-password?
  #:ssl ssl
  #:ssl-context ssl-context
  #:notice-handler notice-handler
  #:notification-handler notification-handler])
→ connection?
 user : string?
 database : string?
 server : string? = "localhost"
 port : exact-positive-integer? = 5432
 socket : (or/c path-string? 'guess #f) = #f
 password : (or/c string? #f) = #f
 allow-cleartext-password? : boolean? = #f
 ssl : (or/c 'yes 'optional 'no) = 'no
 ssl-context : ssl-client-context?
             = (ssl-make-client-context 'sslv3)
 notice-handler : (or/c 'output 'error output-port? = void
                        (-> string? string? any))
 notification-handler : (or/c 'output 'error output-port?
                              (-> string? any)
                              (-> string? string? any))
                      = void
```

Creates a connection to a PostgreSQL server. Only the *database* and *user* arguments are mandatory.

By default, the connection is made via TCP to "localhost" at port 5432. To make a different TCP connection, provide one or both of the *server* and *port* arguments.

To connect via a local socket, specify the socket path as the *socket* argument. You must not supply the *socket* argument if you have also supplied either of the TCP arguments. See also §6.1 "Local Sockets for PostgreSQL and MySQL Servers" for notes on socket paths. Supplying a *socket* argument of 'guess is the same as supplying (postgresql-guess-socket-path). Sockets are only available under Linux (x86) and Mac OS X.

If the server requests password authentication, the *password* argument must be present; otherwise an exception is raised. If the server does not request password authentication, the *password* argument is ignored and may be omitted. A connection normally only sends password hashes (using the md5 authentication method). If the server requests a password sent as cleartext (un-hashed), the connection is aborted unless *allow-cleartext-password?* is true.

If the *ssl* argument is either 'yes or 'optional, the connection attempts to negotiate an SSL connection. If the server refuses SSL, the connection raises an exception if *ssl* was set to 'yes or continues with an unencrypted connection if *ssl* was set to 'optional. By default, SSL provides encryption but does not verify the identity of the server (see this explanation). Host verification can be required via the *ssl-context* argument; see ssl-set-verify!. Some servers use SSL certificates to authenticate clients; see ssl-load-certificate-chain! and ssl-load-private-key!. SSL may only be used with TCP connections, not with local sockets.

The *notice-handler* is called on notice messages received asynchronously from the server. A common example is notice of an index created automatically for a table's primary key. The *notice-handler* function takes two string arguments: the condition's SQLSTATE and a message. The *notification-handler* is called in response to an event notification (see the LISTEN and NOTIFY statements); its arguments are the name of the channel, and the payload, as strings. If the handler only accepts a single argument, then it will be called without the payload. The ability to include a payload in a notification was added in PostgreSQL 9.0. An output port may be supplied instead of a procedure, in which case a message is printed to the given port. Finally, the symbol 'output causes the message to be printed to the current output port, and 'error causes the message to be printed to the current error port.

If the connection cannot be made, an exception is raised.

Examples:

```
> (postgresql-connect #:server "db.mysite.com"
                      #:port 5432
                      #:database "webappdb"
```

```
                          #:user "webapp"
                          #:password "ultra5ecret")
(object:connection% ...)
> (postgresql-connect #:user "me"
                      #:database "me"
                      #:password "icecream")
(object:connection% ...)
> (postgresql-connect ; Typical socket path
                      #:socket "/var/run/postgresql/.s.PGSQL.5432"
                      #:user "me"
                      #:database "me")
(object:connection% ...)
> (postgresql-connect #:socket 'guess ; or (postgresql-guess-
socket-path)
                      #:user "me"
                      #:database "me")
(object:connection% ...)
```

```
(postgresql-guess-socket-path) → path-string?
```

Attempts to guess the path for the socket based on conventional locations. This function returns the first such path that exists in the filesystem. It does not check that the path is a socket file, nor that the path is connected to a PostgreSQL server.

If none of the attempted paths exist, an exception is raised.

```
(mysql-connect  #:user user
               [#:database database
                #:server server
                #:port port
                #:socket socket
                #:ssl ssl
                #:ssl-context ssl-context
                #:password password
                #:notice-handler notice-handler]) → connection?
  user : string?
  database : (or/c string? #f) = #f
  server : string? = "localhost"
  port : exact-positive-integer? = 3306
  socket : (or/c path-string? #f) = #f
  ssl : (or/c 'yes 'optional 'no) = 'no
  ssl-context : ssl-client-context?
              = (ssl-make-client-context 'tls)
  password : (or/c string? #f) = #f
  notice-handler : (or/c 'output 'error output-port?
                         (-> exact-nonnegative-integer? string? any))
                 = void
```

Creates a connection to a MySQL server. If *database* is #f, the connection is established without setting the current database; it should be subsequently set with the USE SQL command.

The meaning of the other keyword arguments is similar to those of the postgresql-connect function, except that the first argument to a *notice-handler* function is a MySQL-specific integer code rather than a SQLSTATE string, and a *socket* argument of 'guess is the same as supplying (mysql-guess-socket-path).

If the connection cannot be made, an exception is raised.

Examples:

```
> (mysql-connect #:server "db.mysite.com"
                 #:port 3306
                 #:database "webappdb"
                 #:user "webapp"
                 #:password "ultra5ecret")
(object:connection% ...)
> (mysql-connect #:user "me"
                 #:database "me"
                 #:password "icecream")
(object:connection% ...)
> (mysql-connect ; Typical socket path
                 #:socket "/var/run/mysqld/mysqld.sock"
                 #:user "me"
                 #:database "me")
(object:connection% ...)
> (mysql-connect #:socket (mysql-guess-socket-path)
                 #:user "me"
                 #:database "me")
(object:connection% ...)
```

```
(mysql-guess-socket-path) → path-string?
```

Attempts to guess the path for the socket based on conventional locations. This function returns the first such path that exists in the filesystem. It does not check that the path is a socket file, nor that the path is connected to a MySQL server.

If none of the attempted paths exist, an exception is raised.

```
(sqlite3-connect  #:database database
                 [#:mode mode
                  #:busy-retry-limit busy-retry-limit
                  #:busy-retry-delay busy-retry-delay
                  #:use-place use-place])
 → connection?
```

```
database : (or/c path-string? 'memory 'temporary)
mode : (or/c 'read-only 'read/write 'create) = 'read/write
busy-retry-limit : (or/c exact-nonnegative-integer? +inf.0)
                 = 10
busy-retry-delay : (and/c rational? (not/c negative?)) = 0.1
use-place : boolean? = #f
```

Opens the SQLite database at the file named by `database`, if `database` is a string or path. If `database` is `'temporary`, a private disk-based database is created. If `database` is `'memory`, a private memory-based database is created.

If `mode` is `'read-only`, the database is opened in read-only mode. If `mode` is `'read/write` (the default), the database is opened for reading and writing (if filesystem permissions permit). The `'create` mode is like `'read/write`, except that if the given file does not exist, it is created as a new database.

SQLite uses coarse-grained locking, and many internal operations fail with the `SQLITE_BUSY` condition when a lock cannot be acquired. When an internal operation fails because the database is busy, the connection sleeps for `busy-retry-delay` seconds and retries the operation, up to `busy-retry-limit` additional times. If `busy-retry-limit` is `0`, the operation is only attempted once. If after `busy-retry-limit` retries the operation still does not succeed, an exception is raised.

If `use-place` is true, the actual connection is created in a distinct place for database connections and a proxy is returned; see §6.7 "FFI-Based Connections and Concurrency".

If the connection cannot be made, an exception is raised.

Examples:

```
> (sqlite3-connect #:database "/path/to/my.db")
(object:connection% ...)
> (sqlite3-connect #:database "relpath/to/my.db"
                   #:mode 'create)
(object:connection% ...)
```

```
(sqlite3-available?) → boolean?
```

Reports whether the SQLite native library is found, in which case `sqlite3-connect` works, otherwise it raises an exception.

```
(odbc-connect
  #:dsn dsn
 [#:user user
  #:password password
  #:notice-handler notice-handler
  #:strict-parameter-types? strict-parameter-types?
  #:character-mode character-mode
  #:use-place use-place])
→ connection?
 dsn : string?
 user : (or/c string? #f) = #f
 password : (or/c string? #f) = #f
 notice-handler : (or/c output-port? 'output 'error = void
                        (-> string? string? any))
 strict-parameter-types? : boolean? = #f
 character-mode : (or/c 'wchar 'utf-8 'latin-1) = 'wchar
 use-place : boolean? = #f
```

Creates a connection to the ODBC Data Source named *dsn*. The *user* and *password* arguments are optional, since that information may be incorporated into the data source definition, or it might not be relevant to the data source's driver. The *notice-handler* argument behaves the same as in `postgresql-connect`.

If *strict-parameter-types?* is true, then the connection attempts to determine and enforce specific types for query parameters. See §4.1.4 "ODBC Types" for more details.

By default, connections use ODBC's `SQL_C_WCHAR`-based character encoding (as UTF-16) to send and receive Unicode character data. Unfortunately, some drivers' support for this method is buggy. To use `SQL_C_CHAR` instead, set *character-mode* to `'utf-8` or `'latin-1`, depending on which encoding the driver uses.

See §6.9 "ODBC Status" for notes on specific ODBC drivers and recommendations for connection options.

If *use-place* is true, the actual connection is created in a distinct place for database connections and a proxy is returned; see §6.7 "FFI-Based Connections and Concurrency".

If the connection cannot be made, an exception is raised.

```
(odbc-driver-connect
  connection-string
 [#:notice-handler notice-handler
  #:strict-parameter-types? strict-parameter-types?
  #:character-mode character-mode
  #:use-place use-place])
→ connection?
 connection-string : string?
```

```
notice-handler : (or/c output-port? 'output 'error = void
                       (-> string? string? any))
strict-parameter-types? : boolean? = #f
character-mode : (or/c 'wchar 'utf-8 'latin-1) = 'wchar
use-place : boolean? = #f
```

Creates a connection using an ODBC connection string containing a sequence of keyword and value connection parameters. The syntax of connection strings is described in SQLDriverConnect (see Comments section); supported attributes depend on the driver. The other arguments are the same as in `odbc-connect`.

If the connection cannot be made, an exception is raised.

```
(odbc-data-sources) → (listof (list/c string? string?))
```

Returns a list of known ODBC Data Sources. Each data souce is represented by a list of two strings; the first string is the name of the data source, and the second is the name of its associated driver.

```
(odbc-drivers) → (listof (cons/c string? any/c))
```

Returns a list of known ODBC Drivers. Each driver is represented by a list, the first element of which is the name of the driver. The contents of the rest of each entry is currently undefined.

2.2 Connection Pooling

Creating a database connection can be a costly operation; it may involve steps such as process creation and SSL negotiation. A *connection pool* helps reduce connection costs by reusing connections.

```
(connection-pool  connect
                 [#:max-connections max-connections
                  #:max-idle-connections max-idle-connections])
 → connection-pool?
  connect : (-> connection?)
  max-connections : (or/c (integer-in 1 10000) +inf.0) = +inf.0
  max-idle-connections : (or/c (integer-in 1 10000) +inf.0) = 10
```

Creates a connection pool. The pool consists of up to *max-connections*, divided between leased connections and up to *max-idle-connections* idle connections. The pool uses *connect* to create new connections when needed. The *connect* function is called with the same `current-custodian` value as when the connection pool was created, and it must return a fresh connection each time it is called.

Examples:

```
> (define pool
    (connection-pool
     (lambda () (displayln "connecting!") (sqlite3-connect ....))
     #:max-idle-connections 1))

> (define c1 (connection-pool-lease pool))
connecting!

> (define c2 (connection-pool-lease pool))
connecting!

> (disconnect c1)

> (define c3 (connection-pool-lease pool)) ; reuses actual conn.
from c1
```

See also `virtual-connection` for a mechanism that eliminates the need to explicitly call `connection-pool-lease` and `disconnect`.

```
(connection-pool? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a connection pool, `#f` otherwise.

```
(connection-pool-lease pool [release]) → connection?
  pool : connection-pool?
  release : (or/c evt? custodian?) = (current-thread)
```

Obtains a connection from the connection pool, using an existing idle connection in *pool* if one is available. If no idle connection is available and the pool contains fewer than its maximum allowed connections, a new connection is created; otherwise an exception is raised.

Calling `disconnect` on the connection obtained causes the connection to be released back to the connection pool. The connection is also released if *release* becomes available, if it is a synchronizable event, or if *release* is shutdown, if it is a custodian. The default for *release* is the current thread, so the resulting connection is released when the thread that requested it terminates.

When a connection is released, it is kept as an idle connection if *pool*'s idle connection limit would not be exceeded; otherwise, it is disconnected. In either case, if the connection is in a transaction, the transaction is rolled back.

2.3 Virtual Connections

A *virtual connection* creates actual connections on demand and automatically releases them when they are no longer needed.

```
(virtual-connection connect) → connection?
  connect : (or/c (-> connection?) connection-pool?)
```

Creates a virtual connection that creates actual connections on demand using the `connect` function, or by calling `(connection-pool-lease connect)` if `connect` is a connection pool. If `connect` is a function, it is called with the same `current-custodian` value as when the virtual connection was created.

A virtual connection encapsulates a mapping of threads to actual connections. When a query function is called with a virtual connection, the current thread's associated actual connection is used to execute the query. If there is no actual connection associated with the current thread, one is obtained by calling `connect`. An actual connection is disconnected when its associated thread dies.

Virtual connections are especially useful in contexts such as web servlets (see §1.4 "Databases and Web Servlets"), where each request is handled in a fresh thread. A single global virtual connection can be defined, freeing each servlet request from explicitly opening and closing its own connections. In particular, a virtual connection backed by a connection pool combines convenience with efficiency:

```
(define the-connection
  (virtual-connection (connection-pool (lambda () ....))))
```

The resulting virtual connection leases a connection from the pool on demand for each servlet request thread and releases it when the thread terminates (that is, when the request has been handled).

When given a connection produced by `virtual-connection`, `connected?` indicates whether there is an actual connection associated with the current thread. Likewise, `disconnect` causes the current actual connection associated with the thread (if there is one) to be disconnected, but the connection will be recreated if a query function is executed.

Examples:

```
> (define c
    (virtual-connection
     (lambda ()
       (printf "connecting!\n")
       (postgresql-connect ....))))
```

```
> (connected? c)
#f
> (query-value c "select 1")
connecting!
1
> (connected? c)
#t
> (void (thread (lambda () (displayln (query-value c "select
2")))))
connecting!
2

> (disconnect c)

> (connected? c)
#f
> (query-value c "select 3")
connecting!
3
```

Connections produced by `virtual-connection` may not be used with the `prepare` function. However, they may still be used to execute parameterized queries expressed as strings or encapsulated via `virtual-statement`.

Examples:

```
> (prepare c "select 2 + $1")
prepare: cannot prepare statement with virtual connection
> (query-value c "select 2 + $1" 2)
4
> (define pst (virtual-statement "select 2 + $1"))

> (query-value c pst 3)
5
```

2.4 Kill-safe Connections

```
(kill-safe-connection c) → connection?
  c : connection?
```

Creates a proxy for connection *c*. All queries performed through the proxy are kill-safe; that is, if a thread is killed during a call to a query function such as `query`, the connection will not become locked or damaged. (Connections are normally thread-safe but not kill-safe.)

Note: A kill-safe connection whose underlying connection uses ports to communicate with a database server is not protected from a custodian shutting down its ports.

2.5 Data Source Names

A DSN (data source name) is a symbol associated with a connection specification in a DSN file. They are inspired by, but distinct from, ODBC's DSNs.

```
(struct data-source (connector args extensions)
    #:mutable)
  connector : (or/c 'postgresql 'mysql 'sqlite3 'odbc)
  args : list?
  extensions : (listof (list/c symbol? any/c))
```

Represents a data source. The `connector` field determines which connection function is used to create the connection. The `args` field is a partial list of arguments passed to the connection function; additional arguments may be added when `dsn-connect` is called. The `extensions` field contains additional information about a connection; for example, this library's testing framework uses it to store SQL dialect flags.

Data sources can also be created using the `postgresql-data-source`, etc auxiliary functions.

```
(dsn-connect  dsn
             [#:dsn-file dsn-file]
              arg ...
              #:<kw> kw-arg ...)   → connection?
  dsn : (or/c symbol? data-source?)
  dsn-file : path-string? = (current-dsn-file)
  arg : any/c
  kw-arg : any/c
```

Makes a connection using the connection information associated with *`dsn`* in *`dsn-file`*. The given *`arg`*s and *`kw-arg`*s are added to those specified by *`dsn`* to form the complete arguments supplied to the connect function.

If *`dsn-file`* does not exist, or if it contains no entry for *`dsn`*, an exception is raised. If *`dsn`* is a `data-source`, then *`dsn-file`* is ignored.

Examples:

```
> (put-dsn 'pg
           (postgresql-data-source #:user "me"
                                   #:database "mydb"
                                   #:password "icecream"))

> (dsn-connect 'pg)
(object:connection% ...)
> (dsn-connect 'pg #:notice-handler (lambda (code msg) ....))
(object:connection% ...)
```

```
(current-dsn-file) → path-string?
(current-dsn-file x) → void?
  x : path-string?
```

A parameter holding the location of the default DSN file. The initial value is a file located immediately within `(find-system-path 'prefs-dir)`.

```
(get-dsn dsn [default #:dsn-file dsn-file])
 → (or/c data-source? any/c)
  dsn : symbol?
  default : any/c = #f
  dsn-file : path-string? = (current-dsn-file)
```

Returns the `data-source` associated with *`dsn`* in *`dsn-file`*.

If *`dsn-file`* does not exist, an exception is raised. If *`dsn-file`* does not have an entry for *`dsn`*, *`default`* is called if it is a function or returned otherwise.

```
(put-dsn dsn ds [#:dsn-file dsn-file]) → void?
  dsn : symbol?
  ds : (or/c data-source? #f)
  dsn-file : path-string? = (current-dsn-file)
```

Associates *`dsn`* with the given data source *`ds`* in *`dsn-file`*, replacing the previous association, if one exists.

```
(postgresql-data-source
 [#:user user
  #:database database
  #:server server
  #:port port
  #:socket socket
  #:password password
  #:allow-cleartext-password? allow-cleartext-password?
  #:ssl ssl
  #:notice-handler notice-handler
  #:notification-handler notification-handler])
→ data-source?
 user : string? = absent
 database : string? = absent
 server : string? = absent
 port : exact-positive-integer? = absent
 socket : (or/c path-string? 'guess #f) = absent
 password : (or/c string? #f) = absent
 allow-cleartext-password? : boolean? = absent
 ssl : (or/c 'yes 'optional 'no) = absent
 notice-handler : (or/c 'output 'error) = absent
 notification-handler : (or/c 'output 'error) = absent
(mysql-data-source [#:user user
                    #:database database
                    #:server server
                    #:port port
                    #:socket socket
                    #:ssl ssl
                    #:password password
                    #:notice-handler notice-handler])
→ data-source?
 user : string? = absent
 database : (or/c string? #f) = absent
 server : string? = absent
 port : exact-positive-integer? = absent
 socket : (or/c path-string? 'guess #f) = absent
 ssl : (or/c 'yes 'optional 'no) = absent
 password : (or/c string? #f) = absent
 notice-handler : (or/c 'output 'error) = absent
```

```
(sqlite3-data-source [#:database database
                      #:mode mode
                      #:busy-retry-limit busy-retry-limit
                      #:busy-retry-delay busy-retry-delay
                      #:use-place use-place])
 → data-source?
  database : (or/c path-string? 'memory 'temporary) = absent
  mode : (or/c 'read-only 'read/write 'create) = absent
  busy-retry-limit : (or/c exact-nonnegative-integer? +inf.0)
                   = absent
  busy-retry-delay : (and/c rational? (not/c negative?))
                   = absent
  use-place : boolean? = absent
```

```
(odbc-data-source
 [#:dsn dsn
  #:database database
  #:user user
  #:password password
  #:notice-handler notice-handler
  #:strict-parameter-types? strict-parameter-types?
  #:character-mode character-mode])
 → data-source?
  dsn : (or/c string? #f) = absent
  database : (or/c string? #f) = absent
  user : (or/c string? #f) = absent
  password : (or/c string? #f) = absent
  notice-handler : (or/c 'output 'error) = absent
  strict-parameter-types? : boolean? = absent
  character-mode : (or/c 'wchar 'utf-8 'latin-1) = absent
```

Analogues of `postgresql-connect`, `mysql-connect`, `sqlite3-connect`, and `odbc-connect`, respectively, that return a `data-source` describing the (partial) connection information. All arguments are optional, even those that are mandatory in the corresponding connection function; the missing arguments must be supplied when `dsn-connect` is called.

2.6 Managing Connections

```
(connection? x) → boolean?
  x : any/c
```

Returns `#t` if `x` is a connection, `#f` otherwise.

```
(disconnect connection) → void?
  connection : connection?
```

Closes the connection.

```
(connected? connection) → boolean?
  connection : connection?
```

Returns `#t` if *connection* is connected, `#f` otherwise.

```
(connection-dbsystem connection) → dbsystem?
  connection : connection?
```

Gets an object encapsulating information about the database system of *connection*.

```
(dbsystem? x) → boolean?
  x : any/c
```

Predicate for objects representing database systems.

```
(dbsystem-name sys) → symbol?
  sys : dbsystem?
```

Returns a symbol that identifies the database system. Currently one of the following:

- `'postgresql`
- `'mysql`
- `'sqlite3`
- `'odbc`

```
(dbsystem-supported-types sys) → (listof symbol?)
  sys : dbsystem?
```

Returns a list of symbols identifying types supported by the database system. See §4.1 "SQL Type Conversions".

2.7 System-specific Modules

The `db` module exports all of the functions listed in this manual except those described in §5 "Utilities". The database system-specific connection modules are loaded lazily to avoid unnecessary dependencies on foreign libraries.

The following modules provide subsets of the bindings described in this manual.

```
(require db/base)
```
package: `db-lib`

Provides all generic connection operations (those described in §2.6 “Managing Connections” and §3 “Queries”) and SQL data support (§4 “SQL Types and Conversions”).

```
(require db/postgresql)
```
package: `db-lib`

Provides only `postgresql-connect` and `postgresql-guess-socket-path`.

```
(require db/mysql)
```
package: `db-lib`

Provides only `mysql-connect` and `mysql-guess-socket-path`.

```
(require db/sqlite3)
```
package: `db-lib`

Provides `sqlite3-connect` plus `sqlite3-available?`. When the SQLite native library cannot be found, `sqlite3-connect` raises an exception.

```
(require db/odbc)
```
package: `db-lib`

Provides only `odbc-connect`, `odbc-driver-connect`, `odbc-data-sources`, and `odbc-drivers`. In contrast to `db`, this module immediately attempts to load the ODBC native library when required, and it raises an exception if it cannot be found.

3 Queries

This library provides a high-level functional query API, unlike many other database libraries, which present a stateful, iteration-based interface to queries. When a query function is invoked, it either returns a result or, if the query caused an error, raises an exception. Different query functions impose different constraints on the query results and offer different mechanisms for processing the results.

Errors In most cases, a query error does not cause the connection to be disconnected. Specifically, the following kinds of errors should never cause a connection to be disconnected:

- SQL syntax errors, such as references to undefined tables, columns, or operations, etc
- SQL runtime errors, such as integrity constraint violations
- violations of a specialized query function's expectations, such as using `query-value` with a query that returns multiple columns
- supplying the wrong number or wrong types of parameters to a prepared query, executing a prepared query with the wrong connection, etc

The following kinds of errors may cause a connection to be disconnected:

- changing communication settings, such as changing the connection's character encoding
- communication failures and internal errors in the library
- a break occurring during a connection operation

See §3.5 "Transactions" for information on how errors can affect the transaction status.

Character encoding This library is designed to interact with database systems using the UTF-8 character encoding. The connection functions attempt to negotiate UTF-8 communication at the beginning of every connection, but some systems also allow the character encoding to be changed via SQL commands (eg, `SET NAMES`). If this happens, the client might be unable to reliably communicate with the database, and data might get corrupted in transmission. Avoid changing a connection's character encoding. When possible, the connection will observe the change and automatically disconnect with an error.

Synchronization Connections are internally synchronized: it is safe to use a connection from different threads concurrently. Most connections are not kill-safe: killing a thread that is using a connection may leave the connection locked, causing future operations to block indefinitely. See also §2.4 "Kill-safe Connections".

3.1 Statements

All query functions require both a connection and a *statement*, which is one of the following:

- a string containing a single SQL statement
- a prepared statement produced by `prepare`
- a virtual statement produced by `virtual-statement`
- a statement-binding value produced by `bind-prepared-statement`
- an instance of a struct type that implements `prop:statement`

A SQL statement may contain parameter placeholders that stand for SQL scalar values; such statements are called *parameterized queries*. The parameter values must be supplied when the statement is executed; the parameterized statement and parameter values are sent to the database back end, which combines them correctly and safely.

Use parameters instead of Racket string operations (eg, `format` or `string-append`) to avoid §1.2.1 "SQL Injection".

The syntax of placeholders varies depending on the database system. For example:

PostgreSQL:	`select * from the_numbers where n > $1;`
MySQL, ODBC:	`select * from the_numbers where n > ?;`
SQLite:	supports both syntaxes (plus others)

```
(statement? x) → boolean?
  x : any/c
```

Returns `#t` if `x` is a statement, `#f` otherwise.

3.2 Simple Queries

The simple query API consists of a set of functions specialized to various types of queries. For example, `query-value` is specialized to queries that return exactly one row of exactly one column.

If a statement takes parameters, the parameter values are given as additional arguments immediately after the SQL statement. Only a statement given as a string, prepared statement, or virtual statement can be given "inline" parameters; if the statement is a statement-binding, no inline parameters are permitted.

The types of parameters and returned fields are described in §4 "SQL Types and Conversions".

```
(query-exec connection stmt arg ...) → void?
  connection : connection?
  stmt : statement?
  arg : any/c
```

Executes a SQL statement for effect.

Examples:

```
> (query-exec pgc "insert into the_numbers values (42, 'the an-
swer')")

> (query-exec pgc "delete from the_numbers where n = $1" 42)
```

```
(query-rows  connection
             stmt
             arg ...
            [#:group groupings
             #:group-mode group-mode]) → (listof vector?)
  connection : connection?
  stmt : statement?
  arg : any/c
  groupings : (let* ([field/c (or/c string? exact-nonnegative-integer?)]
                     [grouping/c (or/c field/c (vectorof field/c))])
                (or/c grouping/c (listof grouping/c)))
            = null
  group-mode : (listof (or/c 'preserve-null 'list)) = null
```

Executes a SQL query, which must produce rows, and returns the list of rows (as vectors) from the query.

Examples:

```
> (query-rows pgc "select * from the_numbers where n = $1" 2)
'(#(2 "company"))
> (query-rows c "select 17")
'(#(17))
```

If *groupings* is not empty, the result is the same as if `group-rows` had been called on the result rows.

```
(query-list connection stmt arg ...) → list?
  connection : connection?
  stmt : statement?
  arg : any/c
```

Executes a SQL query, which must produce rows of exactly one column, and returns the list of values from the query.

Examples:

```
> (query-list c "select n from the_numbers where n < 2")
'(0 1)
> (query-list c "select 'hello'")
'("hello")
```

```
(query-row connection stmt arg ...) → vector?
  connection : connection?
  stmt : statement?
  arg : any/c
```

Executes a SQL query, which must produce exactly one row, and returns its (single) row result as a vector.

Examples:

```
> (query-row pgc "select * from the_numbers where n = $1" 2)
'#(2 "company")
> (query-row pgc "select min(n), max(n) from the_numbers")
'#(0 3)
```

```
(query-maybe-row connection stmt arg ...) → (or/c vector? #f)
  connection : connection?
  stmt : statement?
  arg : any/c
```

Like `query-row`, but the query may produce zero rows; in that case, `#f` is returned.

Examples:

```
> (query-maybe-row pgc "select * from the_numbers where n =
$1" 100)
#f
> (query-maybe-row c "select 17")
'#(17)
```

```
(query-value connection stmt arg ...) → any/c
  connection : connection?
  stmt : statement?
  arg : any/c
```

Executes a SQL query, which must produce exactly one row of exactly one column, and returns its single value result.

Examples:

```
> (query-value pgc "select timestamp 'epoch'")
(sql-timestamp 1970 1 1 0 0 0 0 #f)
> (query-value pgc "select d from the_numbers where n = $1" 3)
"a crowd"
```

```
(query-maybe-value connection stmt arg ...) → (or/c any/c #f)
  connection : connection?
  stmt : statement?
  arg : any/c
```

Like query-value, but the query may produce zero rows; in that case, #f is returned.

Examples:

```
> (query-maybe-value pgc "select d from the_numbers where n =
$1" 100)
#f
> (query-maybe-value c "select count(*) from the_numbers")
4
```

```
(in-query connection
          stmt
          arg ...
         [#:fetch fetch-size
          #:group groupings
          #:group-mode group-mode]) → sequence?
  connection : connection?
  stmt : statement?
  arg : any/c
  fetch-size : (or/c exact-positive-integer? +inf.0) = +inf.0
  groupings : (let* ([field/c (or/c string? exact-nonnegative-integer?)]
                     [grouping/c (or/c field/c (vectorof field/c))])
                (or/c grouping/c (listof grouping/c)))
            = null
  group-mode : (listof (or/c 'preserve-null 'list)) = null
```

Executes a SQL query, which must produce rows, and returns a sequence. Each step in the sequence produces as many values as the rows have columns.

If *fetch-size* is +inf.0, all rows are fetched when the sequence is created. If *fetch-size* is finite, a *cursor* is created and *fetch-size* rows are fetched at a time, allowing processing to be interleaved with retrieval. On some database systems, ending a transaction implicitly closes all open cursors; attempting to fetch more rows may fail. On PostgreSQL, a cursor can be opened only within a transaction.

If *groupings* is not empty, the result is the same as if group-rows had been called on the result rows. If *groupings* is not empty, then *fetch-size* must be +inf.0; otherwise, an exception is raised.

Examples:

```
> (for/list ([n (in-query pgc "select n from the_numbers where n <
2")])
    n)
'(0 1)
> (call-with-transaction pgc
    (lambda ()
      (for ([(n d)
             (in-query pgc "select * from the_numbers where n <
$1" 4
                       #:fetch 1)])
        (printf "~a: ~a\n" n d))))
0: nothing
1: the loneliest number
2: company
3: a crowd
```

An in-query application can provide better performance when it appears directly in a for clause. In addition, it may perform stricter checks on the number of columns returned by the query based on the number of variables in the clause's left-hand side:

Example:

```
> (for ([n (in-query pgc "select * from the_numbers")])
    (displayln n))
in-query: query returned wrong number of columns
  statement: "select * from the_numbers"
  expected: 1
  got: 2
```

3.3 General Query Support

A general query result is either a simple-result or a rows-result.

```
(struct simple-result (info))
  info : (listof (cons/c symbol? any/c))
```

Represents the result of a SQL statement that does not return a relation, such as an INSERT or DELETE statement.

The info field is an association list, but its contents vary based on database system and may change in future versions of this library (even new minor versions). The following keys are supported for multiple database systems:

- 'insert-id: If the value is a positive integer, the statement was an INSERT statement and the value is a system-specific identifier for the inserted row. For PostgreSQL, the value is the row's OID, if the table has OIDs (for an alternative, see the INSERT ... RETURNING statement). For MySQL, the value is the same as the result of last_insert_id function—that is, the value of the row's AUTO_INCREMENT field. If there is no such field, the value is #f. For SQLite, the value is the same as the result of the last_insert_rowid function—that is, the ROWID of the inserted row.
- 'affected-rows: The number (a nonnegative integer) of rows inserted by an INSERT statement, modified by an UPDATE statement, or deleted by a DELETE statement. Only directly affected rows are counted; rows affected because of triggers or integrity constraint actions are not counted.

```
(struct rows-result (headers rows))
  headers : (listof any/c)
  rows : (listof vector?)
```

Represents the result of SQL statement that results in a relation, such as a SELECT query.

The headers field is a list whose length is the number of columns in the result rows. Each header is usually an association list containing information about the column, but do not rely on its contents; it varies based on the database system and may change in future version of this library (even new minor versions).

```
(query connection stmt arg ...)
 → (or/c simple-result? rows-result?)
  connection : connection?
  stmt : statement?
  arg : any/c
```

Executes a query, returning a structure that describes the results. Unlike the more specialized query functions, query supports both rows-returning and effect-only queries.

```
(group-rows  result
             #:group groupings
            [#:group-mode group-mode]) → rows-result?
  result : rows-result?
  groupings : (let* ([field/c (or/c string? exact-nonnegative-integer?)]
                     [grouping/c (or/c field/c (vectorof field/c))])
                (or/c grouping/c (listof grouping/c)))
  group-mode : (listof (or/c 'preserve-null 'list)) = null
```

If *groupings* is a vector, the elements must be names of fields in *result*, and *result*'s rows are regrouped using the given fields. Each grouped row contains N+1 fields; the first N fields are the *groupings*, and the final field is a list of "residual rows" over the rest of the fields. A residual row of all NULLs is dropped (for convenient processing of `OUTER JOIN` results) unless *group-mode* includes `'preserve-null`. If *group-mode* contains `'list`, there must be exactly one residual field, and its values are included without a vector wrapper (similar to `query-list`).

See also §1.3.1 "The N+1 Selects Problem".

Examples:

```
> (define vehicles-result
    (rows-result
     '(((name . "type")) ((name . "maker")) ((name . "model")))
     `(#("car"  "honda"   "civic")
       #("car"  "ford"    "focus")
       #("car"  "ford"    "pinto")
       #("bike" "giant"   "boulder")
       #("bike" "schwinn" ,sql-null))))

> (group-rows vehicles-result
              #:group '(#("type")))
(rows-result
 '(((name . "type"))
   ((name . "grouped") (grouped ((name . "maker")) ((name .
"model")))))
 '(#("car" (#("honda" "civic") #("ford" "focus") #("ford"
"pinto")))
   #("bike" (#("giant" "boulder") #("schwinn" #<sql-null>)))))
```

The grouped final column is given the name `"grouped"`.

The *groupings* argument may also be a list of vectors; in that case, the grouping process is repeated for each set of grouping fields. The grouping fields must be distinct.

Example:

```
> (group-rows vehicles-result
              #:group '(#("type") #("maker"))
              #:group-mode '(list))
(rows-result
 '(((name . "type"))
   ((name . "grouped")
    (grouped
     ((name . "maker"))
     ((name . "grouped") (grouped ((name . "model")))))))
```

```
'(#("car" (#("honda" ("civic")) #("ford" ("focus" "pinto"))))
  #("bike" (#("giant" ("boulder")) #("schwinn" ()))))
```

```
(rows->dict  result
             #:key key-field/s
             #:value value-field/s
            [#:value-mode value-mode]) → dict?
  result : rows-result?
  key-field/s : (let ([field/c (or/c string? exact-nonnegative-integer?)])
                  (or/c field/c (vectorof field/c)))
  value-field/s : (let ([field/c (or/c string? exact-nonnegative-integer?)])
                    (or/c field/c (vectorof field/c)))
  value-mode : (listof (or/c 'list 'preserve-null)) = null
```

Creates a dictionary mapping `key-field/s` to `value-field/s`. If `key-field/s` is a single field name or index, the keys are the field values; if `key-field/s` is a vector, the keys are vectors of the field values. Likewise for `value-field/s`.

If `value-mode` contains `'list`, a list of values is accumulated for each key; otherwise, there must be at most one value for each key. Values consisting of all `sql-null?` values are dropped unless `value-mode` contains `'preserve-null`.

Examples:

```
> (rows->dict vehicles-result
              #:key "model" #:value '#("type" "maker"))
'#hash((#<sql-null> . #("bike" "schwinn"))
       ("civic" . #("car" "honda"))
       ("pinto" . #("car" "ford"))
       ("focus" . #("car" "ford"))
       ("boulder" . #("bike" "giant")))
> (rows->dict vehicles-result
              #:key "maker" #:value "model" #:value-mode '(list))
'#hash(("ford" . ("focus" "pinto"))
       ("honda" . ("civic"))
       ("giant" . ("boulder"))
       ("schwinn" . ()))
```

3.4 Prepared Statements

A *prepared statement* is the result of a call to `prepare`.

Any server-side or native-library resources associated with a prepared statement are released when the prepared statement is garbage-collected or when the connection that owns it is closed; prepared statements do not need to be (and cannot be) explicitly closed.

```
(prepare connection stmt) → prepared-statement?
  connection : connection?
  stmt : (or/c string? virtual-statement?)
```

Prepares a statement. The resulting prepared statement is tied to the connection that prepared it; attempting to execute it with another connection will trigger an exception. The prepared statement holds its connection link weakly; a reference to a prepared statement will not keep a connection from being garbage collected.

```
(prepared-statement? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a prepared statement created by `prepare`, `#f` otherwise.

```
(prepared-statement-parameter-types pst)
 → (listof (list/c boolean? (or/c symbol? #f) any/c))
  pst : prepared-statement?
```

Returns a list with one element for each of the prepared statement's parameters. Each element is itself a list of the following form:

```
(list supported? type typeid)
```

The *supported?* field indicates whether the type is supported by this library; the *type* field is a symbol corresponding to an entry in one of the tables in §4.1 "SQL Type Conversions"; and the *typeid* field is a system-specific type identifier. The type description list format may be extended with additional information in future versions of this library.

```
(prepared-statement-result-types pst)
 → (listof (list/c boolean? (or/c symbol? #f) any/c))
  pst : prepared-statement?
```

If *pst* is a rows-returning statement (eg, SELECT), returns a list of type descriptions as described above, identifying the SQL types (or pseudo-types) of the result columns. If *pst* is not a rows-returning statement, the function returns the empty list.

```
(bind-prepared-statement pst params) → statement-binding?
  pst : prepared-statement?
  params : (listof any/c)
```

Creates a statement-binding value pairing *pst* with *params*, a list of parameter arguments. The result can be executed with `query` or any of the other query functions, but it must be used with the same connection that created *pst*.

Example:

```
> (let* ([get-name-pst
          (prepare pgc "select d from the_numbers where n = $1")]
         [get-name2
          (bind-prepared-statement get-name-pst (list 2))]
         [get-name3
          (bind-prepared-statement get-name-pst (list 3))])
    (list (query-value pgc get-name2)
          (query-value pgc get-name3)))
'("company" "a crowd")
```

Most query functions perform the binding step implicitly.

```
(statement-binding? x) → boolean?
  x : any/c
```

Returns #t if *x* is a statement created by bind-prepared-statement, #f otherwise.

```
(virtual-statement gen) → virtual-statement?
  gen : (or/c string? (-> dbsystem? string?))
```

Creates a *virtual statement*, *stmt*, which encapsulates a weak mapping of connections to prepared statements. When a query function is called with *stmt* and a connection, the weak hash is consulted to see if the statement has already been prepared for that connection. If so, the prepared statement is used; otherwise, the statement is prepared and stored in the table.

The *gen* argument must be either a SQL string or a function that accepts a databse system object and produces a SQL string. The function variant allows the SQL syntax to be dynamically customized for the database system in use.

Examples:

```
> (define pst
    (virtual-statement
     (lambda (dbsys)
       (case (dbsystem-name dbsys)
         ((postgresql) "select n from the_numbers where n < $1")
         ((sqlite3) "select n from the_numbers where n < ?")
         (else (error "unknown system"))))))

> (query-list pgc pst 3)
'(0 1 2)
> (query-list slc pst 3)
'(0 1 2)
```

```
(virtual-statement? x) → boolean?
  x : any/c
```

Returns `#t` if `x` is a virtual statement created by `virtual-statement`, `#f` otherwise.

3.5 Transactions

The functions described in this section provide a consistent interface to transactions.

A *managed transaction* is one created via either `start-transaction` or `call-with-transaction`. In contrast, an *unmanaged transaction* is one created by evaluating a SQL statement such as `START TRANSACTION`. A *nested transaction* is a transaction created within the extent of an existing transaction. If a nested transaction is committed, its changes are promoted to the enclosing transaction, which may itself be committed or rolled back. If a nested transaction is rolled back, its changes are discarded, but the enclosing transaction remains open. Nested transactions are implemented via SQL `SAVEPOINT`, `RELEASE SAVEPOINT`, and `ROLLBACK TO SAVEPOINT`.

ODBC connections must use managed transactions exclusively; using transaction-changing SQL may cause these functions to behave incorrectly and may cause additional problems in the ODBC driver. ODBC connections do not support nested transactions.

PostgreSQL, MySQL, and SQLite connections must not mix managed and unmanaged transactions. For example, calling `start-transaction` and then executing a `ROLLBACK` statement is not allowed. Note that in MySQL, some SQL statements have implicit transaction effects. For example, in MySQL a `CREATE TABLE` statement implicitly commits the current transaction. These statements also must not be used within managed transactions. (In contrast, PostgreSQL and SQLite both support transactional DDL.)

Errors Query errors may affect an open transaction in one of three ways:

1. the transaction remains open and unchanged
2. the transaction is automatically rolled back
3. the transaction becomes an *invalid transaction*; all subsequent queries will fail until the transaction is rolled back

To avoid the silent loss of information, this library attempts to avoid behavior (2) completely by marking transactions as invalid instead (3). Invalid transactions can be identified using the `needs-rollback?` function. The following list is a rough guide to what errors cause which behaviors:

- All errors raised by checks performed by this library, such as parameter arity and type errors, leave the transaction open and unchanged (1).
- All errors originating from PostgreSQL cause the transaction to become invalid (3).

- Most errors originating from MySQL leave the transaction open and unchanged (1), but a few cause the transaction to become invalid (3). In the latter cases, the underlying behavior of MySQL is to roll back the transaction but *leave it open* (see the MySQL documentation). This library detects those cases and marks the transaction invalid instead.
- Most errors originating from SQLite leave the transaction open and unchanged (1), but a few cause the transaction to become invalid (3). In the latter cases, the underlying behavior of SQLite is to roll back the transaction (see the SQLite documentation). This library detects those cases and marks the transaction invalid instead.
- All errors originating from an ODBC driver cause the transaction to become invalid (3). The underlying behavior of ODBC drivers varies widely, and ODBC provides no mechanism to detect when an existing transaction has been rolled back, so this library intercepts all errors and marks the transaction invalid instead.

If a nested transaction marked invalid is rolled back, the enclosing transaction is typically still valid.

If a transaction is open when a connection is disconnected, it is implicitly rolled back.

```
(start-transaction  c
                   [#:isolation isolation-level
                    #:option option])          → void?
  c : connection?
  isolation-level : (or/c 'serializable      = #f
                          'repeatable-read
                          'read-committed
                          'read-uncommitted
                          #f)
  option : any/c = #f
```

Starts a transaction with isolation `isolation-level`. If `isolation-level` is `#f`, the isolation is database-dependent; it may be a default isolation level or it may be the isolation level of the previous transaction.

The behavior of `option` depends on the database system:

- PostgreSQL supports `'read-only` and `'read-write` for the corresponding transaction options.
- SQLite supports `'deferred`, `'immediate`, and `'exclusive` for the corresponding locking modes.
- MySQL and ODBC no not support any options.

If *option* is not supported, an exception is raised.

If *c* is already in a transaction, *isolation-level* and *option* must both be `#f`, and a nested transaction is opened.

See also §1.3.5 "Transactions and Concurrency".

```
(commit-transaction c) → void?
  c : connection?
```

Attempts to commit the current transaction, if one is open. If the transaction cannot be commited (for example, if it is invalid), an exception is raised.

If the current transaction is a nested transaction, the nested transaction is closed, its changes are incorporated into the enclosing transaction, and the enclosing transaction is resumed.

If no transaction is open, this function has no effect.

```
(rollback-transaction c) → void?
  c : connection?
```

Rolls back the current transaction, if one is open.

If the current transaction is a nested transaction, the nested transaction is closed, its changes are abandoned, and the enclosing transaction is resumed.

If no transaction is open, this function has no effect.

```
(in-transaction? c) → boolean?
  c : connection?
```

Returns `#t` if *c* has an open transaction (managed or unmanaged), `#f` otherwise.

```
(needs-rollback? c) → boolean?
  c : connection?
```

Returns `#t` if *c* is in an invalid transaction. All queries executed using *c* will fail until the transaction is rolled back (either using `rollback-transaction`, if the transaction was created with `start-transaction`, or when the procedure passed to `call-with-transaction` returns).

```
(call-with-transaction c
                       proc
                      [#:isolation isolation-level
                       #:option option])          → any
  c : connection?
```

```
proc : (-> any)
isolation-level : (or/c 'serializable      = #f
                        'repeatable-read
                        'read-committed
                        'read-uncommitted
                        #f)
option : any/c = #f
```

Calls *proc* in the context of a new transaction with isolation level *isolation-level*. If *proc* completes normally, the transaction is committed and *proc*'s results are returned. If *proc* raises an exception (or if the implicit commit at the end raises an exception), the transaction is rolled back and the exception is re-raised.

If `call-with-transaction` is called within a transaction, *isolation-level* must be `#f`, and it creates a nested transaction. Within the extent of a call to `call-with-transaction`, transactions must be properly nested. In particular:

- Calling either `commit-transaction` or `rollback-transaction` when the open transaction was created by `call-with-transaction` causes an exception to be raised.
- If a further nested transaction is open when *proc* completes (that is, created by an unmatched `start-transaction` call), an exception is raised and the nested transaction created by `call-with-transaction` is rolled back.

3.6 SQL Errors

SQL errors are represented by the `exn:fail:sql` exception type.

```
(struct exn:fail:sql exn:fail (sqlstate info)
    #:extra-constructor-name make-exn:fail:sql)
  sqlstate : (or/c string? symbol?)
  info : (listof (cons/c symbol? any/c))
```

Represents a SQL error originating from the database server or native library. The `sqlstate` field contains the SQLSTATE code (a five-character string) of the error for PostgreSQL, MySQL, or ODBC connections or a symbol for SQLite connections. Refer to the database system's documentation for the definitions of error codes:

- PostgreSQL SQLSTATE codes
- MySQL SQLSTATE codes

- SQLite error codes; errors are represented as a symbol based on the error constant's name, such as `'busy` for `SQLITE_BUSY`
- ODBC: see the database system's documentation

The `info` field contains all information available about the error as an association list. The available keys vary, but the `'message` key is typically present; its value is a string containing the error message.

Example:

```
> (with-handlers ([exn:fail:sql? exn:fail:sql-info])
    (query pgc "select * from nosuchtable"))
'((severity . "ERROR")
  (code . "42P01")
  (message . "relation \"nosuchtable\" does not exist")
  (position . "15")
  (file . "parse_relation.c")
  (line . "857")
  (routine . "parserOpenTable"))
```

Errors originating from the `db` library, such as arity and contract errors, type conversion errors, etc, are not represented by `exn:fail:sql`.

3.7 Database Catalog Information

```
(list-tables c [#:schema schema]) → (listof string?)
  c : connection?
  schema : (or/c 'search-or-current 'search 'current)
         = 'search-or-current
```

Returns a list of unqualified names of tables (and views) defined in the current database.

If *schema* is `'search`, the list contains all tables in the current schema search path (with the possible exception of system tables); if the search path cannot be determined, an exception is raised. If *schema* is `'current`, the list contains all tables in the current schema. If *schema* is `'search-or-current` (the default), the search path is used if it can be determined; otherwise the current schema is used. The schema search path cannot be determined for ODBC-based connections.

```
(table-exists? c
               table-name
              [#:schema schema
               #:case-sensitive? case-sensitive?]) → boolean?
  c : connection?
```

```
table-name : string?
schema : (or/c 'search-or-current 'search 'current)
       = 'search-or-current
case-sensitive? : any/c = #f
```

Indicates whether a table (or view) named *table-name* exists. The meaning of the *schema* argument is the same as for `list-tables`, and the *case-sensitive?* argument controls how table names are compared.

3.8 Creating New Kinds of Statements

```
prop:statement : (struct-type-property/c
                  (-> any/c connection? statement?))
```

A struct type property for creating new kinds of statements. The property value is applied to the struct instance and a connection, and it must return a statement.

4 SQL Types and Conversions

Connections automatically convert query results to appropriate Racket types. Likewise, query parameters are accepted as Racket values and converted to the appropriate SQL type.

Examples:

```
> (query-value pgc "select count(*) from the_numbers")
4
> (query-value pgc "select false")
#f
> (query-value pgc "select 1 + $1" 2)
3
```

If a query result contains a column with a SQL type not supported by this library, an exception is raised. As a workaround, cast the column to a supported type:

Examples:

```
> (query-value pgc "select inet '127.0.0.1'")
query-value: unsupported type
  type: inet
  typeid: 869
> (query-value pgc "select cast(inet '127.0.0.1' as varchar)")
"127.0.0.1/32"
```

The exception for unsupported types in result columns is raised when the query is executed, not when it is prepared; for parameters it is raised when the parameter values are supplied. Thus even unexecutable prepared statements can be inspected using `prepared-statement-parameter-types` and `prepared-statement-result-types`.

4.1 SQL Type Conversions

This section describes the correspondences between SQL types and Racket types for the supported database systems.

4.1.1 PostgreSQL Types

This section applies to connections created with `postgresql-connect`.

The following table lists the PostgreSQL types known to this library, along with their corresponding Racket representations.

PostgreSQL type	pg_type.typname	Racket type
'boolean	bool	boolean?
'char1	char	char?
'smallint	int2	exact-integer?
'integer	int4	exact-integer?
'bigint	int8	exact-integer?
'real	float4	real?
'double	float8	real?
'decimal	numeric	rational? or +nan.0
'character	bpchar	string?
'varchar	varchar	string?
'text	text	string?
'bytea	bytea	bytes?
'date	date	sql-date?
'time	time	sql-time?
'timetz	timetz	sql-time?
'timestamp	timestamp	sql-timestamp? or -inf.0 or +inf.0
'timestamptz	timestamptz	sql-timestamp? or -inf.0 or +inf.0
'interval	interval	sql-interval?
'bit	bit	bit-vector?
'varbit	varbit	bit-vector?
'json	json	jsexpr?
'jsonb	jsonb	jsexpr?
'int4range	int4range	pg-range-or-empty?
'int8range	int8range	pg-range-or-empty?
'numrange	numrange	pg-range-or-empty?
'tsrange	tsrange	pg-range-or-empty?
'tstzrange	tstzrange	pg-range-or-empty?
'daterange	daterange	pg-range-or-empty?
'point	point	point?
'lseg	lseg	line?
'path	path	pg-path?
'box	box	pg-box?
'polygon	polygon	polygon?
'circle	circle	pg-circle?

The 'char1 type, written "char" in PostgreSQL's SQL syntax (the quotation marks are significant), is one byte, essentially a tiny integer written as a character.

A SQL value of type decimal is converted to either an exact rational or +nan.0. When converting Racket values to SQL decimal, exact rational values representable by finite decimal strings are converted without loss of precision. (Precision may be lost, of course, if the value is then stored in a database field of lower precision.) Other real values are converted to decimals with a loss of precision. In PostgreSQL, numeric and decimal refer to the same type.

Examples:

```
> (query-value pgc "select real '+Infinity'")
+inf.0
> (query-value pgc "select numeric '12345678901234567890'")
12345678901234567890
```

A SQL `timestamp with time zone` is converted to a Racket `sql-timestamp` in UTC—that is, with a `tz` field of `0`. If a Racket `sql-timestamp` without a time zone (`tz` is `#f`) is given for a parameter of type `timestamp with time zone`, it is treated as a timestamp in UTC. See also §6.4 "PostgreSQL Timestamps and Time Zones".

The geometric types such as `'point` are represented by structures defined in the `db/util/geometry` and `db/util/postgresql` modules.

PostgreSQL user-defined *domains* are supported in query results if the underlying type is supported. Recordset headers and `prepared-statement-result-types` report them in terms of the underlying type. Parameters with user-defined domain types are not currently supported. As a workaround, cast the parameter to the underlying type. For example, if the type of `$1` is a domain whose underlying type is `integer`, then replace `$1` with `($1::integer)`.

For each type in the table above, the corresponding array type is also supported, using the `pg-array` structure. Use the `= ANY` syntax with an array parameter instead of dynamically constructing a SQL `IN` expression:

Examples:

```
> (query-value pgc "select 1 in (1, 2, 3)")
#t
> (query-value pgc "select 1 = any ($1::integer[])"
               (list->pg-array (list 1 2 3)))
#t
```

A list may be provided for an array parameter, in which case it is automatically converted using `list->pg-array`. The type annotation can be dropped when the array type can be inferred from the left-hand side.

Examples:

```
> (query-value pgc "select 1 = any ($1)" (list 1 2 3))
#t
> (query-value pgc "select $1::integer = any ($2)"
               1 (list 1 2 3))
#t
> (query-value pgc "select $1 = any ($2)" ; what type are we
using?
               1 (list 1 2 3))
```

query-value: cannot convert given value to SQL type
given: 1
type: string
expected: string?
dialect: PostgreSQL

PostgreSQL defines many other types, such as network addresses and row types. These are currently not supported, but support may be added in future versions of this library.

4.1.2 MySQL Types

This section applies to connections created with `mysql-connect`.

The following table lists the MySQL types known to this library, along with their corresponding Racket representations.

MySQL type	Racket type
`'integer`	`exact-integer?`
`'tinyint`	`exact-integer?`
`'smallint`	`exact-integer?`
`'mediumint`	`exact-integer?`
`'bigint`	`exact-integer?`
`'real`	`real?`
`'double`	`real?`
`'decimal`	`exact?`
`'varchar`	`string?`
`'date`	`sql-date?`
`'time`	`sql-time?` or `sql-day-time-interval?`
`'datetime`	`sql-timestamp?`
`'var-string`	`string?`
`'text`	`string?`
`'var-binary`	`bytes?`
`'blob`	`bytes?`
`'bit`	`bit-vector?`
`'geometry`	`geometry2d?`

MySQL does not report specific parameter types for prepared queries, so they are instead assigned the pseudo-type `'any`. Conversion of Racket values to parameters accepts strings, numbers (`rational?`—no infinities or NaN), bytes, SQL date/time structures (`sql-date?`, `sql-time?`, `sql-timestamp?`, and `sql-day-time-interval?`), bits (`bit-vector?`), and geometric values (`geometry2d?`). Numbers are sent as 64-bit signed integers, if possible, or as double-precision floating point numbers otherwise.

Fields of type `CHAR` or `VARCHAR` are typically reported as `'var-string`, and fields of type `BINARY` or `VARBINARY` are typically reported as `'var-binary`.

The MySQL `time` type represents time intervals, which may not correspond to times of day (for example, the interval may be negative or larger than 24 hours). In conversion from MySQL results to Racket values, those `time` values that represent times of day are converted to `sql-time` values; the rest are represented by `sql-interval` values.

The MySQL `enum` and `set` types are not supported. As a workaround, cast them to/from either integers or strings.

4.1.3 SQLite Types

This section applies to connections created with `sqlite3-connect`.

The following table lists the SQLite types known to this library, along with their corresponding Racket representations.

Unlike PostgreSQL and MySQL, SQLite does not enforce declared type constraints (with the exception of `integer primary key`) on *columns*. Rather, every SQLite *value* has an associated "storage class".

SQLite storage class	Racket type
`integer`	`exact-integer?`
`real`	`real?`
`text`	`string?`
`blob`	`bytes?`

SQLite does not report specific parameter and result types for prepared queries. Instead, they are assigned the pseudo-type `'any`. Conversion of Racket values to parameters accepts strings, bytes, and real numbers.

An exact integer that cannot be represented as a 64-bit signed integer is converted as `real`, not `integer`.

Examples:

```
> (expt 2 80)
1208925819614629174706176
> (query-value slc "select ?" (expt 2 80))
1.2089258196146292e+24
```

4.1.4 ODBC Types

This section applies to connections created with `odbc-connect` or `odbc-driver-connect`.

The following table lists the ODBC types known to this library, along with their corresponding Racket representations.

ODBC type	**Racket type**
'character	string?
'varchar	string?
'longvarchar	string?
'numeric	rational?
'decimal	rational?
'integer	exact-integer?
'tinyint	exact-integer?
'smallint	exact-integer?
'bigint	exact-integer?
'float	real?
'real	real?
'double	real?
'date	sql-date?
'time	sql-time?
'datetime	sql-timestamp?
'timestamp	sql-timestamp?
'binary	bytes?
'varbinary	bytes?
'longvarbinary	bytes?
'bit1	boolean?

Not all ODBC drivers provide specific parameter type information for prepared queries. Some omit parameter type information entirely or, worse, assign all parameters a single type such as varchar. To avoid enforcing irrelevant type constraints in the last case, connections only attempt to fetch and enforce parameter types when the connection is made using the #:strict-parameter-type? option. Otherwise, the connection assigns all parameters the type 'unknown. (The 'unknown type is also used when specific parameter types are requested but are not available.) Conversion of Racket values to 'unknown parameters accepts strings, bytes, numbers (rational?—no infinities or NaN), booleans, and SQL date/time structures (sql-date?, sql-time?, and sql-timestamp?).

The ODBC type 'bit1 represents a single bit, unlike the standard SQL bit(N) type.

Interval types are not currently supported on ODBC.

4.2 SQL Data

This section describes data types for representing various SQL types that have no existing appropriate counterpart in Racket.

4.2.1 SQL NULL

SQL `NULL` is translated into the unique `sql-null` value.

```
sql-null : sql-null?
```

A special value used to represent `NULL` values in query results. The `sql-null` value may be recognized using `eq?`.

Example:

```
> (query-value pgc "select NULL")
#<sql-null>
```

```
(sql-null? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is `sql-null`; `#f` otherwise.

```
(sql-null->false x) → any/c
  x : any/c
```

Returns `#f` if *x* is `sql-null`; otherwise returns *x*.

Examples:

```
> (sql-null->false "apple")
"apple"
> (sql-null->false sql-null)
#f
> (sql-null->false #f)
#f
```

```
(false->sql-null x) → any/c
  x : any/c
```

Returns `sql-null` if *x* is `#f`; otherwise returns *x*.

Examples:

```
> (false->sql-null "apple")
"apple"
> (false->sql-null #f)
#<sql-null>
```

4.2.2 Dates and Times

The DATE, TIME (WITH TIME ZONE and without), TIMESTAMP (WITH TIME ZONE and without), and INTERVAL SQL types are represented by the following structures.

See also §5.1 "Datetime Type Utilities" for more functions on datetime values.

```
(struct sql-date (year month day))
  year : exact-integer?
  month : (integer-in 0 12)
  day : (integer-in 0 31)
```

Represents a SQL date.

Dates with zero-valued month or day components are a MySQL extension.

```
(struct sql-time (hour minute second nanosecond tz))
  hour : exact-nonnegative-integer?
  minute : exact-nonnegative-integer?
  second : exact-nonnegative-integer?
  nanosecond : exact-nonnegative-integer?
  tz : (or/c exact-integer? #f)
```

```
(struct sql-timestamp (year
                       month
                       day
                       hour
                       minute
                       second
                       nanosecond
                       tz))
  year : exact-nonnegative-integer?
  month : exact-nonnegative-integer?
  day : exact-nonnegative-integer?
  hour : exact-nonnegative-integer?
  minute : exact-nonnegative-integer?
  second : exact-nonnegative-integer?
  nanosecond : exact-nonnegative-integer?
  tz : (or/c exact-integer? #f)
```

Represents SQL times and timestamps.

The tz field indicates the time zone offset as the number of seconds east of GMT (as in srfi/19). If tz is #f, the time or timestamp does not carry time zone information.

The sql-time and sql-timestamp structures store fractional seconds to nanosecond precision for compatibility with srfi/19. Note, however, that database systems generally do not support nanosecond precision; PostgreSQL, for example, only supports microsecond precision.

Examples:

```
> (query-value pgc "select date '25-dec-1980'")
(sql-date 1980 12 25)
> (query-value pgc "select time '7:30'")
(sql-time 7 30 0 0 #f)
> (query-value pgc "select timestamp 'epoch'")
(sql-timestamp 1970 1 1 0 0 0 0 #f)
> (query-value pgc "select timestamp with time zone 'epoch'")
(sql-timestamp 1970 1 1 0 0 0 0 0)
```

```
(struct sql-interval (years
                      months
                      days
                      hours
                      minutes
                      seconds
                      nanoseconds))
  years : exact-integer?
  months : exact-integer?
  days : exact-integer?
  hours : exact-integer?
  minutes : exact-integer?
  seconds : exact-integer?
  nanoseconds : exact-integer?
```

Represents lengths of time. Intervals are normalized to satisfy the following constraints:

- years and months have the same sign
- months ranges from -11 to 11
- days, hours, minutes, seconds, and nanoseconds all have the same sign
- hours ranges from -23 to 23
- minutes and seconds range from -59 to 59
- nanoseconds ranges from (- (sub1 #e1e9)) to (sub1 #e1e9)

That is, an interval consists of two groups of components: year-month and day-time, and normalization is done only within groups. In fact, the SQL standard recognizes those two types of intervals separately (see `sql-year-month-interval?` and `sql-day-time-interval?`, below), and does not permit combining them. Intervals such as `1 month 3 days` are a PostgreSQL extension.

```
(sql-year-month-interval? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a `sql-interval` value where the `days`, `hours`, `minutes`, `seconds`, and `nanoseconds` fields are zero.

```
(sql-day-time-interval? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a `sql-interval` value where the `years` and `months` fields are zero.

```
(sql-interval->sql-time interval [failure]) → any
  interval : sql-interval?
  failure : any/c = (lambda () (error ....))
```

If *interval* is a day-time interval that represents a time of day, returns the corresponding `sql-time` value. In particular, the following must be true:

- `hours`, `minutes`, `seconds`, and `nanoseconds` must all be non-negative
- `hours` must be between `0` and `23`

The corresponding constraints on `minutes`, etc are enforced by the constructor.

If *interval* is out of range, the *failure* value is called, if it is a procedure, or returned, otherwise.

```
(sql-time->sql-interval time) → sql-day-time-interval?
  time : sql-time?
```

Converts *time* to an interval. If *time* has time-zone information, it is ignored.

4.2.3 Bits

The `BIT` and `BIT VARYING` (`VARBIT`) SQL types are represented by bit-vectors (`data/bit-vector`).

The following functions are provided for backwards compatibility. They are deprecated and will be removed in a future release of Racket.

```
(make-sql-bits len) → sql-bits?
  len : exact-nonnegative-integer?
(sql-bits? v) → boolean?
  v : any/c
(sql-bits-length b) → exact-nonnegative-integer?
  b : sql-bits?
(sql-bits-ref b i) → boolean?
  b : sql-bits?
  i : exact-nonnegative-integer?
(sql-bits-set! b i v) → void?
  b : sql-bits?
  i : exact-nonnegative-integer?
  v : boolean?
(sql-bits->list b) → (listof boolean?)
  b : sql-bits?
(sql-bits->string b) → string?
  b : sql-bits?
(list->sql-bits lst) → sql-bits?
  lst : (listof boolean?)
(string->sql-bits s) → sql-bits?
  s : string?
```

Deprecated; use `data/bit-vector` instead.

5 Utilities

The bindings described in this section are provided by the specific modules below, not by `db` or `db/base`.

5.1 Datetime Type Utilities

```
(require db/util/datetime)        package: db-lib
```

```
(sql-datetime->srfi-date t) → srfi:date?
  t : (or/c sql-date? sql-time? sql-timestamp?)
(srfi-date->sql-date d) → sql-date?
  d : srfi:date?
(srfi-date->sql-time d) → sql-time?
  d : srfi:date?
(srfi-date->sql-time-tz d) → sql-time?
  d : srfi:date?
(srfi-date->sql-timestamp d) → sql-timestamp?
  d : srfi:date?
(srfi-date->sql-timestamp-tz d) → sql-timestamp?
  d : srfi:date?
```

Converts between this library's date and time values and SRFI 19's date values (see `srfi/19`). SRFI dates store more information than SQL dates and times, so converting a SQL time to a SRFI date, for example, puts zeroes in the year, month, and day fields.

Examples:

```
> (sql-datetime->srfi-date
    (query-value pgc "select time '7:30'"))
date-year-day: TIME-ERROR type invalid-month-specification:
0
> (sql-datetime->srfi-date
    (query-value pgc "select date '25-dec-1980'"))
eval: unable to replay evaluation of
(sql-datetime->srfi-date (query-value pgc "select date
'25-dec-1980'"))
> (sql-datetime->srfi-date
    (query-value pgc "select timestamp 'epoch'"))
eval: unable to replay evaluation of
(sql-datetime->srfi-date (query-value pgc "select timestamp
'epoch'"))
```

```
(sql-day-time-interval->seconds interval) → rational?
  interval : sql-day-time-interval?
```

Returns the length of *interval* in seconds.

5.2 Geometric Types

```
(require db/util/geometry)        package: db-lib
```

The following structures and functions deal with geometric values based on the OpenGIS (ISO 19125) model.

Note: Geometric columns defined using the PostGIS extension to PostgreSQL are not directly supported. Instead, data should be exchanged in the Well-Known Binary format; conversion of the following structures to and from WKB format is supported by the `wkb->geometry` and `geometry->wkb` functions.

```
(struct point (x y))
  x : real?
  y : real?
```

Represents an OpenGIS `Point`.

```
(struct line-string (points))
  points : (listof point?)
```

Represents an OpenGIS `LineString`.

```
(struct polygon (exterior interior))
  exterior : linear-ring?
  interior : (listof linear-ring?)
```

Represents an OpenGIS `Polygon`.

```
(struct multi-point (elements))
  elements : (listof point?)
```

Represents an OpenGIS `MultiPoint`, a collection of points.

```
(struct multi-line-string (elements))
  elements : (listof line-string?)
```

Represents an OpenGIS `MultiLineString`, a collection of line-strings.

```
(struct multi-polygon (elements))
  elements : (listof polygon?)
```

Represents an OpenGIS `MultiPolygon`, a collection of polygons.

```
(struct geometry-collection (elements))
  elements : (listof geometry2d?)
```

Represents an OpenGIS `GeometryCollection`, a collection of arbitrary geometric values.

```
(geometry2d? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a `point`, `line-string`, `polygon`, `multi-point`, `multi-line-string`, `multi-polygon`, or `geometry-collection`; `#f` othewise.

```
(line? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a `line-string` consisting of exactly two points (cf OpenGIS `Line`); `#f` otherwise.

```
(linear-ring? x) → boolean?
  x : any/c
```

Returns `#t` if *x* is a `line-string` whose first and last points are equal (cf OpenGIS `LinearRing`); `#f` otherwise.

```
(geometry->wkb g #:big-endian? big-endian?) → bytes?
  g : geometry2d?
  big-endian? : (system-big-endian?)
```

Returns the Well-Known Binary (WKB) encoding of the geometric value *g*. The *big-endian?* argument determines the byte order used (the WKB format includes byte-order markers, so a robust client should accept either encoding).

```
(wkb->geometry b) → geometry2d?
  b : bytes?
```

Decodes the Well-Known Binary (WKB) representation of a geometric value.

5.3 PostgreSQL-specific Types

```
(require db/util/postgresql)      package: db-lib
```

```
(struct pg-array (dimensions
                  dimension-lengths
                  dimension-lower-bounds
                  contents))
  dimensions : exact-nonnegative-integer?
  dimension-lengths : (listof exact-positive-integer?)
  dimension-lower-bounds : (listof exact-integer?)
  contents : vector?
```

Represents a PostrgreSQL array. The `dimension-lengths` and `dimension-lower-bounds` fields are both lists of `dimensions` elements. By default, PostgreSQL array indexes start with 1 (not 0), so `dimension-lower-bounds` is typically a list of `1`s.

```
(pg-array-ref arr index ...+) → any/c
  arr : pg-array?
  index : exact-integer?
```

Returns the element of *arr* at the given position. There must be as many *index* arguments as the dimension of *arr*. Recall that PostgreSQL array indexes usually start with 1, not 0.

```
(pg-array->list arr) → list?
  arr : pg-array?
```

Returns a list of *arr*'s contents. The dimension of *arr* must be 1; otherwise an error is raised.

```
(list->pg-array lst) → pg-array?
  lst : list?
```

Returns a `pg-array` of dimension 1 with the contents of *lst*.

```
(struct pg-empty-range ())
```

Represents an empty range.

```
(struct pg-range (lb includes-lb? ub includes-ub?))
  lb : range-type
  includes-lb? : boolean?
  ub : range-type
  includes-ub? : boolean?
```

Represents a range of values from `lb` (lower bound) to `ub` (upper bound). The `includes-lb?` and `includes-ub?` fields indicate whether each end of the range is open or closed.

The `lb` and `ub` fields must have the same type; the permissible types are exact integers, real numbers, and `sql-timestamp`s. Either or both bounds may also be `#f`, which indicates the range is unbounded on that end.

```
(pg-range-or-empty? v) → boolean?
  v : any/c
```

Returns `#t` if *v* is a `pg-range` or `pg-empty-range` instance; otherwise, returns `#f`.

```
(struct pg-box (ne sw))
  ne : point?
  sw : point?
(struct pg-path (closed? points))
  closed? : boolean?
  points : (listof point?)
(struct pg-circle (center radius))
  center : point?
  radius : real?
```

These structures represent certain of PostgreSQL's built-in geometric types that have no appropriate analogue in the OpenGIS model: `box`, `path`, and `circle`. The `point`, `lseg`, and `polygon` PostgreSQL built-in types are represented using `point`, `line-string` (`line?`), and `polygon` structures.

Note: PostgreSQL's built-in geometric types are distinct from those provided by the PostGIS extension library (see §5.2 "Geometric Types").

5.4 Testing Database Programs

```
(require db/util/testing)        package: db-lib
```

This module provides utilities for testing programs that use database connections.

```
(high-latency-connection  connection
                          latency
                         [#:sleep-atomic? sleep-atomic?])
 → connection?
  connection : connection?
  latency : (>=/c 0)
  sleep-atomic? : any/c = #f
```

Returns a proxy connection for *connection* that introduces *latency* additional seconds of latency before operations that require communicating with the database back end—`prepare`, `query`, `start-transaction`, etc.

Use this function in performance testing to roughly simulate environments with high-latency communication with a database back end.

If *sleep-atomic?* is true, then the proxy enters atomic mode before sleeping, to better simulate the effect of a long-running FFI call (see §6.7 "FFI-Based Connections and Concurrency"). Even so, it may not accurately simulate an ODBC connection that internally uses cursors to fetch data on demand, as each fetch would introduce additional latency.

6 Notes

This section discusses issues related to specific database systems.

6.1 Local Sockets for PostgreSQL and MySQL Servers

PostgreSQL and MySQL servers are sometimes configured by default to listen only on local sockets (also called "unix domain sockets"). This library provides support for communication over local sockets on Linux (x86 and x86-64) and Mac OS X. If local socket communication is not available, the server must be reconfigured to listen on a TCP port.

The socket file for a PostgreSQL server is located in the directory specified by the `unix_socket_directory` variable in the `postgresql.conf` server configuration file. For example, on Ubuntu 11.04 running PostgreSQL 8.4, the socket directory is `/var/run/postgresql` and the socket file is `/var/run/postgresql/.s.PGSQL.5432`. Common socket paths may be searched automatically using the `postgresql-guess-socket-path` function.

The socket file for a MySQL server is located at the path specified by the `socket` variable in the `my.cnf` configuration file. For example, on Ubuntu 11.04 running MySQL 5.1, the socket is located at `/var/run/mysqld/mysqld.sock`. Common socket paths for MySQL can be searched using the `mysql-guess-socket-path` function.

6.2 PostgreSQL Database Character Encoding

In most cases, a database's character encoding is irrelevant, since the connect function always requests translation to Unicode (UTF-8) when creating a connection. If a PostgreSQL database's character encoding is `SQL_ASCII`, however, PostgreSQL will not honor the connection encoding; it will instead send untranslated octets, which will cause corrupt data or internal errors in the client connection.

To convert a PostgreSQL database from `SQL_ASCII` to something sensible, `pg_dump` the database, recode the dump file (using a utility such as `iconv`), create a new database with the desired encoding, and `pg_restore` from the recoded dump file.

6.3 PostgreSQL Authentication

PostgreSQL supports a large variety of authentication mechanisms, controlled by the `pg_hba.conf` server configuration file. This library currently supports only cleartext and md5-hashed passwords, and it does not send cleartext passwords unless explicitly ordered to (see `postgresql-connect`). These correspond to the `md5` and `password` authentication methods in the parlance of pg_hba.conf, respectively. On Linux and Mac OS X, ident authentication is automatically supported for local sockets (as of PostgreSQL 9.1, this authentication method has been renamed peer). The gss, sspi, krb5, pam, and ldap methods are not supported.

6.4 PostgreSQL Timestamps and Time Zones

PostgreSQL's timestamp with time zone type is inconsistent with the SQL standard (probably), inconsistent with time with time zone, and potentially confusing to PostgreSQL newcomers.

A time with time zone is essentially a time structure with an additional field storing a time zone offset. In contrast, a timestamp with time zone has no fields beyond those of timestamp. Rather, it indicates that its datetime fields should be interpreted as a UTC time. Thus it represents an absolute point in time, unlike timestamp without time zone, which represents local date and time in some unknown time zone (possibly—hopefully—known the the database designer, but unknown to PostgreSQL).

When a timestamp with time zone is created from a source without time zone information, the session's TIME ZONE setting is used to adjust the source to UTC time. When the source contains time zone information, it is used to adjust the timestamp to UTC time. In either case, the time zone information is *discarded* and only the UTC timestamp is stored. When a timestamp with time zone is rendered as text, it is first adjusted to the time zone specified by the TIME ZONE setting (or by AT TIME ZONE) and that offset is included in the rendered text.

This library receives timestamps in binary format, so the time zone adjustment is not applied, nor is the session's TIME ZONE offset included; thus all sql-timestamp values in a query result have a tz field of 0 (for timestamp with time zone) or #f (for timestamp without time zone). (Previous versions of this library sent and received timestamps as text, so they received timestamps with adjusted time zones.)

6.5 MySQL Authentication

As of version 5.5.7, MySQL supports authentication plugins. The only plugins currently supported by this library are mysql_native_password (the default) and mysql_old_password, which corresponds to the password authentication mechanisms used since version 4.1 and before 4.1, respectively.

6.6 SQLite Requirements

SQLite support requires the appropriate native library.

- On Windows, the library is `sqlite3.dll`. It is included in the Racket distribution.
- On Mac OS X, the library is `libsqlite3.0.dylib`, which is included (in `/usr/lib`) in Mac OS X version 10.4 onwards.
- On Linux, the library is `libsqlite3.so.0`. It is included in the `libsqlite3-0` package in Debian/Ubuntu and in the `sqlite` package in Red Hat.

6.7 FFI-Based Connections and Concurrency

Wire-based connections communicate using ports, which do not cause other Racket threads to block. In contrast, an FFI call causes all Racket threads to block until it completes, so FFI-based connections can degrade the interactivity of a Racket program, particularly if long-running queries are performed using the connection. This problem can be avoided by creating the FFI-based connection in a separate place using the `#:use-place` keyword argument. Such a connection will not block all Racket threads during queries; the disadvantage is the cost of creating and communicating with a separate place.

6.8 ODBC Requirements

ODBC requires the appropriate driver manager native library as well as driver native libraries for each database system you want use ODBC to connect to.

- On Windows, the driver manager is `odbc32.dll`, which is included automatically with Windows.
- On Mac OS X, the driver manager is `libiodbc.2.dylib` (iODBC), which is included (in `/usr/lib`) in Mac OS X version 10.2 onwards.
- On Linux, the driver manager is `libodbc.so.1` (unixODBC—iODBC is not supported). It is available from the `unixodbc` package in Debian/Ubuntu and in the `unixODBC` package in Red Hat.

In addition, you must install the appropriate ODBC Drivers and configure Data Sources. Refer to the ODBC documentation for the specific database system for more information.

6.9 ODBC Status

ODBC support is experimental. The behavior of ODBC connections can vary widely depending on the driver in use and even the configuration of a particular data source.

The following sections describe the configurations that this library has been tested with.

Reports of success or failure on other platforms or with other drivers would be appreciated.

6.9.1 DB2 ODBC Driver

The driver from IBM DB2 Express-C v9.7 has been tested on Ubuntu 11.04 (32-bit only).

For a typical installation where the instance resides at `/home/db2inst1`, set the following option in the Driver configuration: `Driver = /home/db2inst1/sqllib/lib32/libdb2.so`. (The path would presumably be different for a 64-bit installation.)

The DB2 driver does not seem to accept a separate argument for the database to connect to; it must be the same as the Data Source name.

6.9.2 Oracle ODBC Driver

The driver from Oracle Database 10g Release 2 Express Edition has been tested on Ubuntu 11.04 (32-bit only).

It seems the `ORACLE_HOME` and `LD_LIBRARY_PATH` environment variables must be set according to the `oracle_env.{csh,sh}` script for the driver to work.

Columns of type `TIME` can cause a memory error (ie, Racket crashes). This seems to be due to a bug in Oracle's ODBC driver, but I do not yet have a workaround.

6.9.3 SQL Server ODBC Driver

Basic SQL Server support has been verified on Windows, but the automated test suite has not yet been adapted and run.

The "SQL Server" driver refuses to accept `NUMERIC` or `DECIMAL` parameters, producing the error "Invalid precision value (SQLSTATE: HY104)." If possible, use the "Native SQL Server" driver instead.