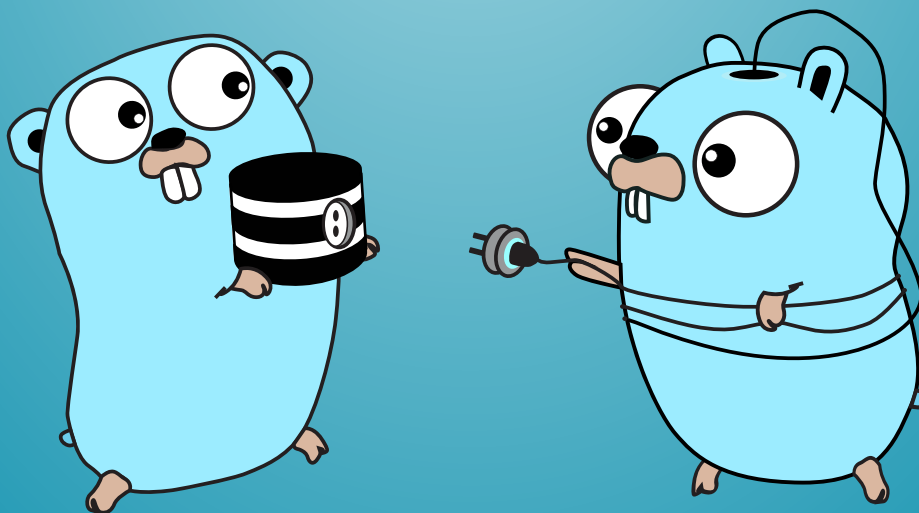


The Ultimate Guide To Building Database - Driven Apps with Go



Abstract

The `database/sql` package is Go's lightweight standard interface for relational databases, with support for backend drivers, prepared statements and transactions. Although it is not complicated, it is different from database interfaces in other languages, and learning to use it properly takes time. This eBook contains years of collected wisdom from experts and is the fastest way to learn how to connect to a database from Go programs. After reading this book, you'll know how to use `database/sql` properly, avoiding many mistakes and ensuring high performance and correct code.

Table of Contents

● Introduction	3
● What is <code>database/sql</code> ?	4
● Your First <code>database/sql</code> Program	6
● Using a <code>sql.DB</code>	8
● Fetching Result Sets	12
● Modifying Data With <code>db.Exec()</code>	21
● Using Prepared Statements	23
● Working With Transactions	30
● Error Handling	34
● Using Built-In Interfaces	36
● Database Drivers	40
● Common Pitfalls	41
● Conclusion	43

Introduction

Congratulations! You've discovered the ultimate resource for writing database-driven applications in the Go programming language. Using Go to access databases brings you all the benefits of Go itself, plus an elegant database interface and a vibrant community of users and developers writing high-quality, open-source database drivers for you to use.

What is Go, and why would you use it? Go is a modern language in the C family. It is elegant, simple, and clear to write and read, which makes it maintainable. It includes a garbage collector to manage memory for you. Its built-in features make it easy to write concurrent programs. These include goroutines, which you can think of as lightweight threads, and mechanisms to communicate amongst goroutines. At the same time, Go is strongly typed and compiles to self-contained binaries free of external dependencies, and is high-performance and efficient in terms of CPU and memory usage.

Go is an excellent choice for systems programming, where you might otherwise choose Java, C or C++ for performance reasons. Typical tasks are building API servers, web servers and other high-performance networked systems, system utilities, databases, and micro-services architectures (APIs and services).

Go is also very popular for tasks where you would otherwise use dynamic scripting languages such as Python and Ruby, which give you simplicity, clarity and flexibility but not high performance. Go gives you many of the best features of these languages, and some properties not present in any of them.

Along with all of these benefits, Go includes a standard library of code for tasks such as encryption, networking, filesystem access, and database access. The database access library is called **database/sql**, and like the rest of Go, is elegant and minimal, with just enough batteries included. It does heavy lifting and repetitive tasks for you, such as connection pooling and retries on errors. But it doesn't bury its internals in abstractions, so your code remains explicit and magic-free.

Go's `database/sql` library has excellent documentation and clear source code but leaves a great deal of learning to the user. Fortunately, you've found this book, which will save you a huge amount of time and mistakes! Contained within this book is years of collected wisdom from many experienced programmers, distilled to just what you need to know, when you need to know it.

A note: as of the time of writing, Go 1.4 has just been released. If that's an old version by the time you read this, you should be aware that there may be changes.

Congratulations on choosing Go and `database/sql`, and on finding this book. Let's get started right away!

What is `database/sql`?

`database/sql` is a package of functionality that's included in Go's standard library. It is the idiomatic, official way to communicate with a row-oriented database. Loosely speaking, it is designed for databases that are relational, or similar to relational databases (many non-relational databases work just fine with it too).

`database/sql` provides much the same functionality for Go that you'd find in ODBC, Perl's DBI, Java's JDBC, and similar. However, it is not designed exactly the same as those, and you should be careful not to assume your knowledge of other database interfaces applies directly in Go.

The `database/sql` package handles non-database-specific aspects of database communication. These are tasks that have to be handled in common across many databases, so they're factored out into a uniform interface for you to use. Database-specific functionality is provided by drivers, which aren't part of the standard library. Many excellent drivers are available in open-source form, and we will discuss those later.

To a large extent, `database/sql` is database-agnostic. The benefits of using it are that your code will be as decoupled from the underlying

database as possible, enabling easier portability and ensuring your code doesn't get cluttered. This leads to better maintainability and understandability.

The `database/sql` package is very idiomatic and Go-ish, following the Go philosophy of not hiding things behind too much abstraction. As a programmer, you'll be in direct control over resource management, including memory management (although, like other things in Go, that is a very small burden due to the language design).



The package follows the Go philosophy of not hiding things behind abstraction. As a programmer, you'll be in direct control over resource management.

What is it not? The primary thing you should be aware of is that it is definitely not an ORM (object-relational mapper) or other similar abstraction. Go, as a language, isn't really oriented towards the type of programming that ORMs provide, and although there are some third-party libraries that attempt to provide ORM-like functionality and convenience helpers (e.g. populating structs with rows from the database), all of them fall short for various reasons.

The `database/sql` package provides several types for you, each of which embodies a concept or set of concepts:

DB. The `sql.DB` type represents a database. Unlike in many other programming languages, it doesn't represent a connection to a database, but rather the database as an object you can manipulate. Connections are managed in an internal connection pool and are not exposed to you directly. This allows you to use databases that are actually connectionless, such as shared-memory or embedded databases, through the same abstraction without worrying about exactly how you communicate with them.

Results. There are several data types that embody the results of database interactions: a `sql.Rows` for fetching multi-row results from a query, a `sql.Row` for a single-row result, and `sql.Result` for examining the effects of statements that modify the database.

Statements. A `sql.Stmt` represents a statement such as DDL, DML, and the like. You can interact with them directly as prepared statements, or indirectly by using convenience functions on the `sql.DB` variable itself.

Transactions. A `sql.Tx` represents a transaction with specific properties, in exchange for bypassing many of the usual conveniences such as the connection pool.

We'll see how to use all these data types, and the abstractions they present, in the following sections. Let's get started with a quick-start: a "hello, world" program!

Your First database/sql Program

This section presents a quick introduction to the major functionality of `database/sql` in the form of a fully functioning Go program! Before you begin, ensure you have access to a MySQL database, as we'll use MySQL for our examples in this book. If you don't have an instance of MySQL that's appropriate for testing, you can get one in seconds with the [MySQL Sandbox](#) utility.

Create a new Go source file, `hello_mysql.go`, with the following source code ([download](#)). You may need to adjust the connection parameters as needed to connect to your testing database. Note also that the example assumes the default test database exists and your user has rights to it:

```
package main

import (
    "database/sql"
    "log"
    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:@tcp(:3306)/test")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    _, err = db.Exec(
        "CREATE TABLE IF NOT EXISTS test.hello(world varchar(50))")
    if err != nil {
        log.Fatal(err)
    }
}
```

```

res, err := db.Exec(
    "INSERT INTO test.hello(world) VALUES('hello world!')")
if err != nil {
    log.Fatal(err)
}
rowCount, err := res.RowsAffected()
if err != nil {
    log.Fatal(err)
}
log.Printf("inserted %d rows", rowCount)

rows, err := db.Query("SELECT * FROM test.hello")
if err != nil {
    log.Fatal(err)
}
for rows.Next() {
    var s string
    err = rows.Scan(&s)
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("found row containing %q", s)
}
rows.Close()
}

```

Run your new Go program with `go run hello_mysql.go`. It's safe to run it multiple times. As you do, you should see output like the following as it continues to insert rows into the table:

```

Desktop $ go run hello_mysql.go
2014/12/16 10:57:03 inserted 1 rows
2014/12/16 10:57:03 found row containing "hello world!"

Desktop $ go run hello_mysql.go
2014/12/16 10:57:05 inserted 1 rows
2014/12/16 10:57:05 found row containing "hello world!"
2014/12/16 10:57:05 found row containing "hello world!"

Desktop $ go run hello_mysql.go
2014/12/16 10:57:07 inserted 1 rows
2014/12/16 10:57:07 found row containing "hello world!"
2014/12/16 10:57:07 found row containing "hello world!"
2014/12/16 10:57:07 found row containing "hello world!"

```

Congratulations! You've written your first program to interact with a MySQL server using Go. What might surprise you is that the code you've just run is not some kind of overly simplified, silly example. It is very similar to the code you'll use in production systems under high load, including error handling. We'll explore much of this code in further sections of this book and learn more about what it does. For now, we'll just mention a few highlights:

- You imported `database/sql` and loaded a driver for MySQL.
- You created a `sql.DB` with a call to `sql.Open()`, passing the driver name and the connection string.
- You used `Exec()` to create a table and insert a row, then inspect the results.
- You used `Query()` to select the rows from the table, `rows.Next()` to iterate over them, and `rows.Scan()` to copy columns from the current row into variables.
- You used `.Close()` to clean up resources when you finished with them.

Let's dig into each of these topics, and more, in detail. We'll begin with the `sql.DB` itself and see what it is and how it works.

Using a `sql.DB`

As mentioned previously, a `sql.DB` is an abstraction of a database. (A common mistake is to think of it as a connection to a database.) It exposes a set of functions you use to interact with the database. Internally, it manages a connection pool for you (a very important topic in this book), and handles a great deal of tedious and repetitive work for you, all in a way that's safe to use concurrently from multiple goroutines.

The intent of a `sql.DB` is that you'll create one object to represent each database you'll use, and keep it for a long time. Because it

has a connection pool, it's not meant to be created and destroyed continually. You should make your single `sql.DB` available to all the code that needs it.

To create a `sql.DB`, as we saw in the previous section, you need to import a driver. We'll cover drivers in more detail later. For now it's enough to note that drivers are usually imported for side effects only. That's why we bind their import name to the `_` anonymous variable, so their namespace isn't usable from your code. All told, the necessary imports you need in your program's main file are as follows:

```
import (
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
)
```

Again we're using our favorite MySQL driver as an example.

You only need to import the driver once in one file, typically `main.go` or the equivalent (if you use some custom wrapper around database functionality, you'd likely import the driver inside that wrapper library). From here on, you'll interact with the `database/sql` types, and it will interact with the library on your behalf, so you don't need access to the driver directly. Behind the scenes, drivers use an `init()` function to register themselves with the `database/sql` package.

Now, to actually create an instance of a `sql.DB`, you use `sql.Open()` with two arguments. The first is the driver's name. This is the string that the driver registers with `database/sql`, and is typically the same as its package name to avoid confusion.

The second argument is the connection string, or DSN (data source name) as some people call it. This is driver-specific and has no meaning to `database/sql`. It is merely passed to the driver you identify. It might include a TCP connection endpoint, a Unix socket, username and password, a filename, or anything else you can think of. Check the driver's documentation for details.



A `sql.DB` is not a connection to a database. It's an abstraction that represents the database itself.

How The Connection Pool Works

At this point you might think you have a connection to a database, but you don't. You have only created an object and associated it with a driver. The `database/sql` package relies on the driver to create and manage individual connections, and keeps a pool of them. The pool is initially empty, and connections are created lazily when needed. The connection pool is vitally important to understand because it affects your program's behavior greatly. That's why we are including details about it early in this book.

The way the pool works is fairly simple in concept. When you call a function that requires access to the underlying database, the function first asks for a connection from the pool. If there's a free one, the pool gives it to the function. Otherwise it opens a new one. The connection is then owned by the function. When the function completes, it either returns the connection to the pool, or passes ownership of the connection to an object, which will release it back to the pool when it is finished.

The specific functions you can call and how they're handled follow, assuming a `sql.DB` variable named `db`:

- `db.Ping()` returns the connection to the pool immediately.
- `db.Exec()` returns the connection to the pool immediately, but the returned `Result` object has a reference to the connection, so it may be used later for inspecting the results of the `Exec()`.
- `db.Query()` passes ownership of the connection to a `sql.Rows` object, which releases it back to the pool when you've fully iterated all the rows or when `.Close()` is called.
- `db.QueryRow()` passes the connection to a `sql.Row`, which releases it when `.Scan()` is called.
- `db.Begin()` passes the connection to a `sql.Tx`, which releases it when `.Commit()` or `.Rollback()` is called.

Given that every connection is lazy-loaded as needed, how are you to validate that you can really use your `sql.DB` after creating it with `sql.Open()`? That is what `db.Ping()` is for. Idiomatic code looks like this:

```
db, err := sql.Open("driverName", "dataSourceName")
if err != nil {
    log.Fatal(err)
}
defer db.Close()
err = db.Ping()
if err != nil {
    log.Fatal(err)
}
```

Now you know that your `db` variable is really ready to use, because `Ping()` has created a single connection and returned it to the pool.

The above code, by the way, is not really idiomatic in one regard. Usually you'd do something more intelligent than just fatally logging an error. But in this book we'll always show `log.Fatal(err)` as a placeholder for real error handling.

A consequence of the connection pool is that you do not need to check for or attempt to handle connection failures. If a connection fails when you perform an operation on the database, `database/sql` will take care of it for you. Internally, it retries up to 10 times when a connection in the pool is discovered to be dead. It simply fetches another from the pool or opens a new one. This means that your code can be clean and free of messy retry logic.

Configuring the Connection Pool

Early versions of Go didn't offer much control over the connection pool, but in Go version 1.2.1 and later, there are options to control it. (There was a bug in version 1.2's connection pool; use at least version 1.2.1). These are as follows:



Understanding the built-in connection pooling and hard-coded 10 automatic retries is key to learning how to use database/sql well.

- `db.SetMaxOpenConns(n int)`. This sets the maximum number of connections the pool will open to the database. This includes connections that are in-use as well as connections that are idle in the pool. If you make a call that requests a connection from the pool, and there isn't a free one and the limit is reached, then your call will block, potentially for a long time. The default limit is 0, which means unlimited.
- `db.SetMaxIdleConns(n int)` sets the number of connections that will be kept idle in the pool after being released. The default is 0, which means that connections are not kept idle in the pool at all: they are closed when released from service. This can lead to a lot of connections being closed and opened rapidly, which is probably not what you want.

The key things to notice about the connection pool are that, depending on how you use connections and how you've configured the pool, it's possible to have a few undesired behaviors:

1. Lots of connection thrash, leading to extra work and latency.
2. Too many connections open to the database, leading to errors.
3. Blocking while waiting for a connection.
4. Operations can fail if the pool has 10 or more dead connections, due to the built-in limit of 10 retries.

Most of the time, how you use the `sql.DB` influences these behaviors more than how you configure the pool. We'll explore this throughout this book. For now, let's move on to our next topic, fetching results from the database and doing useful things with them!

Fetching Result Sets

The `database/sql` library provides specific functions intended for queries that return results: `db.Query()` and `db.QueryRow()`. We've already seen an example of the former, and we'll cover the latter in this section as well.

As described previously, executing `db.Query()` with a SQL query will do the following:

1. Get a connection from the pool
2. Execute the query
3. Transfer ownership of the connection to the result set

The result set, a `sql.Rows` variable that is traditionally called `rows` if no more descriptive name is needed, is then a cursor over the results. Each row is fetched with a call to `rows.Next()`, beginning with the first one. The cursor is initially positioned before the first row.

To repeat the earlier example:

```
rows, err := db.Query("SELECT * FROM test.hello")
if err != nil {
    log.Fatal(err)
}
for rows.Next() {
    var s string
    err = rows.Scan(&s)
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("found row containing %q", s)
}
rows.Close()
```

There are a few things to know about this code, and we'll examine it outside-in, beginning with iterating over the rows with `rows.Next()`.

Iterating Over Rows In A Result

The `rows.Next()` function is designed for use in a `for` loop as shown. When it encounters an error, including `io.EOF` which signals the end of the rows has been reached, it will return `false`. In normal operation, you'll usually iterate over all the rows until the last one, which will exit the loop.

But what if you don't exit the loop normally? What if you intentionally break out of it or return from the function? If this happens, your results won't be fetched and processed completely, and the

connection might not be released back to the pool. Handling rows correctly requires thinking about this possibility. Your goal should be that `rows.Close()` is always called to release its connection back to the pool. If it isn't, then the connection will never go back into the pool, and this can cause serious problems manifested as a "leakage" of connections. If you're not careful you can easily cause server problems or reach your database server's maximum number of connections, causing downtime or unavailability.

How do you prevent this? First, you'll be happy to know that if the loop terminates due to `rows.Next()` returning false, whether normally or abnormally, `rows.Close()` is automatically called for you, so in normal operation you won't reserve connections from the pool in these cases.

The remaining cases are an early return or breaking out of the loop. What you should do in these cases depends on the circumstances. If you will return from the enclosing function when processing ends, you should use `defer rows.Close()`. This is the idiomatic way to ensure that "must-run" code indeed always runs when the function returns. And it's also idiomatic (and important for correctness) to place such a cleanup call immediately after the resource is created. Our modified code would then look like this:

```
rows, err := db.Query("SELECT * FROM test.hello")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
```

However, if the enclosing function is long-lived and you're repeatedly querying in a loop, then you should not defer closing the rows. You should do it explicitly just before breaking out of the loop. In fact, as a general rule, you should call `rows.Close()` as early as you possibly can, to free the resources as soon as possible. This may require a little thought and analysis of your code, in more complex cases.

There are multiple reasons not to defer in a long-lived function:

You should call `rows.Close()` as early as you possibly can, to free the resources as soon as possible.

1. The deferred code won't execute for a potentially long time. You need it to execute ASAP to clean up its resources right away.
2. The deferred function, and the variables it refers to, consumes memory. If the function is really long-lived then this is a memory leak.

With result set cleanup behind us, let's look at handling cases where the result set's loop exits abnormally. We've seen that the normal reason for it to exit is when the loop encounters an `io.EOF` error, making `rows.Next()` return false. Anytime `rows.Next()` finds an error, it saves the error internally for later inspection, and exits the loop. You can then examine it with `rows.Err()`. We didn't show this in the examples above, but in real production code you should always check for an error after exiting the loop:

```
for rows.Next() {  
    // process the rows  
}  
  
if err = rows.Err(); err != nil {  
    log.Fatal(err)  
}
```

The `io.EOF` error is a special case that is handled inside `rows.Err()`. You do not need to handle this explicitly in your code; `rows.Err()` will just return `nil` so you won't see it.

That pretty much concludes everything you need to know about looping over the rows, except for one small detail: handling errors from `rows.Close()`. Interestingly, this function does return an error, but it's a good question what can be done with it. If it doesn't make sense for your code to handle it (and we haven't seen a case where it does), then you can feel free to ignore it or just log it and continue.

Fetching A Single Row

Fetching a single row is a very common task that's awkward with the code shown previously. You'd have to write a loop, check that

the loop actually had some rows, and so forth. Fortunately, there's `db.QueryRow()` that can do this for you. It executes a query that's expected to return zero or one rows, and returns a `sql.Row` object that is scannable. The usual idiom is to chain the query and scan together, like this:

```
var s string
err = db.QueryRow("select * from hello.world limit 1").Scan(&s)
if err != nil {
    if err == sql.ErrNoRows {
        // special case: there was no row
    } else {
        log.Fatal(err)
    }
}
log.Println("found a row", s)
```



Simplify your code by letting `Scan()` handle type conversion for you.

As you can see, the idiomatic usage is a little different from before. Internally, the `sql.Row` object holds either an error from the query, or a `sql.Rows` from the query. If there's an error, then `.Scan()` will return the deferred error. If there's none, then `.Scan()` will work as usual, except that if there was no row, it returns a special error constant, `sql.ErrNoRows`. You can check for this error to determine whether the call to `.Scan()` actually executed and copied values from the row into your destination variables.

How rows.Scan() Works

Using `rows.Scan()` and its single-row variant is actually a fairly involved subject. Under the hood, it does quite a bit of work for you. If you know what it's doing, you can use it to great effect.

The arguments to `rows.Scan()` are destinations into which the columns from the row should be stored. Often these will be straightforward pointers to variables, dereferenced with the `&` operator:

```
var var1, var2 string
err = rows.Scan(&var1, &var2)
```


The argument types are the empty interface, `interface{}`, which as you know is satisfied by any type in Go. In most cases, Go copies data from the row into the destinations you give. There are special cases where the copy can be avoided if you want, but you have to use `*sql.RawBytes` types to do that, and the memory is owned by the database and has a limited lifetime of validity. If you need this behavior, you should read the documentation and the source code to learn more about how it works, but most people won't need it. In common usage, you will end up with a copy of the data, which you own and can use as you please. Note that you can't use `*sql.RawBytes` with `db.QueryRow().Scan()` due to an internal limitation in `database/sql`.

The `database/sql` package will examine the type of the destination, and will convert values in many cases. This can help make your code, especially your error-handling code, much smaller. For example, suppose we have a column of numbers, but for some reason it's not numeric, it is instead a `VARCHAR` with numbers in ASCII format. We could scan the column into a string variable, convert it into a number, and check errors at each step. But we don't need to, because `database/sql` can do it for us. If we pass, say, a `float64` destination variable into the call, `Scan()` will detect that we are trying to scan a string into a number, call `strconv.ParseFloat()` for us, and return any errors.

Another special case with scanning is when values are `NULL` in the database. A `NULL` can't be scanned into an ordinary variable, and you can't pass a `nil` into `rows.Scan()`. Instead, you must use a special type as the scan destination. These types are defined in `database/sql` for many common types, such as `sql.NullFloat64` and so forth. If you need a type that isn't defined, you can look to see whether your driver provides one, or copy/paste the source code to make your own; it's only a few lines of code. After scanning, you can check whether the value was valid or not, and get the value if it was. (If you don't care, you can just skip the validity check; reading the value will give the underlying type's zero-value.)

Putting it all together, a more complex call to `rows.Scan()` might look like the following:

```

var (
    s1 string
    s2 sql.NullString
    i1 int
    f1 float64
    f2 float64
)
// Suppose the row contains ["hello", NULL, 12345, "12345.6789", "not-a-float"]
err = rows.Scan(&s1, &s2, &i1, &f1, &f2)
if err != nil {
    log.Fatal(err)
}

```

The call to `rows.Scan()` will fail with the following error, illustrating that the last column was automatically converted to a float, but it failed:

```

sql: Scan error on column index 4: converting string "not-a-float" to
a float64: strconv.ParseFloat: parsing "not-a-float": invalid syntax

```

However, since the arguments are handled in order, the rest of the scans would have succeeded, and by removing the call to `log.Fatal()` we can see that with the following lines of code:

```

err = rows.Scan(&s1, &s2, &i1, &f1, &f2)
log.Printf("%q %#v %d %f %f", s1, s2, i1, f1, f2)

```

The result is:

```

"hello" sql.NullString{String:"", Valid:false} 12345 12345.678900
0.000000

```

This illustrates that the `s2` variable's `Valid` field is false and its `String` field is empty, as expected. Your code could inspect this variable and handle that as desired:

```

if s2.Valid {
    // use s2.String
}

```

What If You Don't Know The Columns?

Sometimes you're querying something that might return an unknown number of columns with unknown names and types. Imagine that you're doing a **SELECT *** inside a backup program, for example. Or perhaps you're querying something that has different columns in different versions of the server, such as **SHOW FULL PROCESSLIST** in MySQL.

The `database/sql` package provides a way to get the column names, and therefore the number of columns, but doesn't directly provide a way to get their types. To get the column names, use `rows.Columns()`. It returns an error, so check that:

```
cols, err := rows.Columns()
if err != nil {
    log.Fatal(err)
}
```

Now you can do something useful with the results. In the simplest case, when you know the columns and types but you expect a variable number of them to be used in different scenarios, you can write something like the following code. Suppose that you get at most 5 columns but in some cases fewer, and they are of type `uint64`, `string`, `string`, `string`, `uint32`. Define a slice of `interface{}` with valid variables (not `nil` pointers) that handles the largest case, then pass the appropriate sized slice of that to `Scan()`:

```
dest := []interface{}{
    new(uint64),
    new(string),
    new(string),
    new(string),
    new(uint32),
}
err = rows.Scan(dest[:len(cols)])
```

If you don't know the columns or the data types, you need to resort to `sql.RawBytes`.

```
cols, err := rows.Columns()
vals := make([]interface{}, len(cols))
for i, _ := range cols {
    vals[i] = new(sql.RawBytes)
}
for rows.Next() {
    err = rows.Scan(vals...)
}
```

After scanning, you can examine the `vals` slice. For each element you should check whether it's `nil`, and use type introspection and type assertions to figure out the type of the variable and handle it. The resulting code is usually not very pretty, but when dealing with unknown data that's about the best you can do.

Working With Multiple Result Sets And Multiple Statements

The `database/sql` package doesn't have any functionality to cope with a query that returns more than one result set. This can range from a non-issue to a showstopper. A lot depends on the database and the driver implementation, but `database/sql` itself is designed for a query to return a single result set, and is not capable of fetching the next result set or handling changes in columns after the first row is fetched.

In MySQL, at least, this also makes it awkward to call stored procedures, even those that don't return multiple result sets. The reason is that MySQL's network protocol enters multi-statement mode for this scenario even when only a single result set will be returned, and if you execute a `CALL XXX.YYY` statement you'll get the following error:

```
Error 1312: PROCEDURE XXX.YYY can't return a result set in the
given context.
```

Similarly, some databases allow multiple statements to be sent in a single query, perhaps semicolon-delimited or the like.

But `database/sql` is not built for this, so the resulting behavior is undefined. For example, the following may execute one or both statements, or just throw an error, depending on the driver and database.

```
_, err := db.Exec("DELETE FROM tb11; DELETE FROM tb12")
```

Modifying Data With `db.Exec()`

Thus far you've been working mostly with `db.Query()` and `db.QueryRow()`, but you've seen `db.Exec()` in action a few times. This is the method you should use for statements that don't return rows. Here's an example:

```
res, err := db.Exec("DELETE FROM hello.world LIMIT 1")
if err != nil {
    log.Fatal(err)
}
rowCnt, err := res.RowsAffected()
if err != nil {
    log.Fatal(err)
}
log.Println("deleted rows:", rowCnt)
```

The `db.Exec()` call returns a `sql.Result`, which you can use to get the number of rows affected, as shown. You can also use it to fetch the auto-increment ID of the last-inserted row, although support for that varies by driver and database. In PostgreSQL, for example, you should use `INSERT RETURNING` and `db.QueryRow()` to fetch the desired value as a result set.



You should always use Exec() for statements that do not return rows.

There's a vitally important difference between `db.Exec()` and `db.Query()`, and it isn't just a matter of being pedantic. As mentioned earlier in this book, `db.Exec()` releases its connection back to the pool right away, whereas `db.Query()` keeps it out of the pool until `rows.Close()` is called. The following code ignores the returned `rows`, and will cause problems:

```
_ , err := db.Query("DELETE FROM hello.world LIMIT 1")
```

The problem is that although the first value returned from the method is assigned to the `_` variable and is inaccessible to the program after that, it's still really assigned, with all the usual consequences. And it won't go away until it's garbage collected. Worse, the connection that's bound to it will *never* be returned to the connection pool. This is a good way to "leak" connections and run the server out of available connections.

In addition to the above, there are some more subtleties you should know about the `Result`. Go guarantees that the database connection that was used to create the `Result` is the same one used for `LastInsertId()` and `RowsAffected()`, and that it's taken out of the pool for these operations and locked. But beyond that, it's an interface type, and the exact behavior will be dependent on the underlying database and the driver's provided implementation.

For example:

- MySQL can use a **BIGINT UNSIGNED** as an auto-increment column, so it's possible for the last-inserted row's column to be too large to fit in `int64`, the returned type defined by `LastInsertId()`.
- The MySQL driver we prefer doesn't make an extra round-trip to the database to find out the last-inserted value and number of rows affected. This information is returned from the server in the wire protocol, and stored in a struct, so there's no need for it. (The connection is still taken out of the pool and locked, then put back, even though it's not used. This is done by `database/sql`, not the driver. So even though this function doesn't access the database, it may still block waiting if its connection is busy.)

- Whether `LastInsertId()` and `RowsAffected()` return errors is driver-specific. In the MySQL driver, for example, they will never return an error. You should not rely on driver-specific details like this, though. Adhere to the contract that is publicly promised in the `database/sql` interface: functions that return errors should be checked for errors.
- Some of the behavior of these functions varies between databases or implementations. For example, suppose a database driver provides `RowsAffected()` but implements it by making a query to the underlying database (e.g. needlessly calling `SELECT LAST_INSERT_ID()` in MySQL instead of using the values returned in the protocol). As mentioned, the original connection will be used, so to that extent the behavior will be correct, but what if other work has been done on that connection in the meantime? You'd be subject to a race condition. This is an area where you'll need to know the actual implementation you're working with.

In general, the database drivers we've seen and worked with are well implemented and you don't need to worry about these finer points most of the time. But we want you to be aware of the details regardless.

Now let's see how to use prepared statements!

Using Prepared Statements

Although we haven't discussed or shown it yet, `database/sql` is rather heavily oriented towards prepared statements.

What is a prepared statement? It is a statement that's sent in a partially-completed form to the server, with placeholders for values that will be filled in later. Now the statement is ready to be executed, hence the name "prepared."

The meaning of "prepared" is system-dependent, but typically the server evaluates the skeleton statement for validity and makes sure it'll be OK to execute. For example, it will usually check that all of the

databases and tables mentioned exist and the user has privileges to access them, and may also partially plan the query's execution. The server then sends back a statement identifier, which is typically bound to the specific connection that was used to prepare the statement (e.g. the statement is not valid for other connections). The statement can then be executed multiple times by sending a special command with the statement identifier and any parameters to be used.



Prepared statements can be confusing sometimes, because some languages' drivers and database interfaces emulate this behavior but don't really use server-side prepared statements per se. For example, in the Perl DBI, the default behavior with the MySQL driver is to show what looks like prepared statements to the programmer, but use so-called "client-side prepared statements" instead (which are really nothing of the sort). In practice this basically means that the DBI interface is concatenating strings and quoting them before sending the full SQL to the server, and prepared statements don't enter the picture at all. Various database abstraction layers in many languages do similar things, usually without really making it visible to the programmer. If this isn't confusing enough, there are even more confusing scenarios, such as MySQL's so-called SQL interface to prepared statements, which has flummoxed many otherwise smart people (including driver authors) into doing strange things.



The `database/sql` package uses prepared statements transparently. By knowing how it works, you'll be able to control it as you wish.

Go's `database/sql` handles prepared statements as first-class citizens, and has a `sql.Stmt` type for them. In fact, `database/sql` prefers to use prepared statements, and a lot of the interface you've seen thus far in this book will use them if you just add parameters to your method calls. For example, let's add a parameter and placeholder to our `INSERT` from earlier:


```
res, err := db.Exec(  
    "INSERT INTO test.hello(world) VALUES(?)", "hello world!")
```

See what we did there? We removed the literal `hello world!` from the SQL and put a `?` placeholder in its place, then called the method with the value as a parameter. Under the hood, Go handles this as follows:

1. It treats parameter 0 as the statement, and prepares it with the server.
2. It executes the resulting prepared statement with the rest of the parameters (in this case just one).
3. It closes the prepared statement.

Prepared statements have their benefits:

- They are convenient; they help you avoid code to quote and assemble SQL.
- They guard against SQL injection and other attacks by avoiding quoting vulnerabilities, so they enhance security.
- There may be driver-specific, protocol-specific, and database-specific performance and other enhancements. For example, in MySQL, prepared statements use the so-called binary protocol for sending parameters to the server, which can be more efficient than sending them as ASCII.
- They can reward you with additional efficiency by eliminating repeated SQL parsing, execution plan generation, and so on.

Some of these benefits apply no matter how many times statements are executed, but some are only beneficial if a statement will be repeatedly re-executed. As a result, you should be aware of the automatic use of prepared statements when you call functions such as `db.Query()` and `db.Exec()` with more than one parameter.

In addition to behind-the-scenes use of prepared statements, you

can prepare statements explicitly. In fact, to reuse them and gain some of the benefits mentioned above, you *must* prepare them explicitly. Use the `db.Prepare()` function for that. The result is a `sql.Stmt` variable:

```
stmt, err := db.Prepare("INSERT INTO test.hello(world) VALUES(?)")
if err != nil {
    log.Fatal(err)
}
```

Now you can repeatedly execute the statement with parameters as desired. The `stmt` variable's method signatures match those you've been using thus far on the `db` variable. In this case we'll use `stmt.Exec()`:

```
for _, str := range []string{"hello1", "hello2", "hello3"} {
    res, err := stmt.Exec(str)
    if err != nil {
        log.Fatal(err)
    }
}
```

A couple of anti-patterns with prepared statements arise fairly often:

- Single-use prepared statements. This is potentially wasteful unless you really want to do it for some reason, e.g. the convenience of avoiding quoting and SQL concatenation yourself. Be aware that every one-off prepared statement at least triples the number of network round-trips you actually make to the backend database: prepare, execute, and close. As you'll see in the next section, it can be even worse than this.
- Re-preparing in a loop. This is just a magnification of the previous point. Make sure you prepare outside the loop, and execute inside of it!

Although we didn't show it in our sample code above, it's a good idea to close prepared statements when you're done with them. You can just call `stmt.Close()` when you are finished, or you can

defer `stmt.Close()`. Either way, keep in mind the same types of considerations we discussed previously with `rows.Close()`.

The Relationship Between Statements And Connections

Because `database/sql` handles a connection pool for you without exposing you to connections directly, the relationship between prepared statements and connections also has to be managed for you behind the scenes. This is worth knowing about, because it has consequences for performance and resource utilization, especially at high concurrency.

When you prepare a statement with `db.Prepare()` and get a `stmt` in return, what really happens is that the statement is prepared on some connection in the pool, but the connection is then released back to the pool. The statement remembers the connection it used. When you execute the statement, it tries to get that connection, but if it's busy, it will re-prepare the statement on another connection, adding this connection to the list of remembered statements. If you re-execute the statement again and all of the connections on which it was previously prepared are busy, it'll prepare the statement on yet another connection, and so on.

So what's really happening behind the scenes is that a given statement might be prepared on many different connections, and thus from the database server's point of view, the number of prepared statements may be much larger than the number of `sql.Stmt` variables you've created in your code.

This situation happens most under high load, when lots of connections are busy, leading to lots of re-preparing. In the worst cases, the authors have seen statements appear to "leak" due to being re-prepared as many times as there are connections. When combined with bugs that lead to connections not being returned to the pool as previously discussed, it's even possible to exceed the maximum number of statements the server will permit to be prepared at one time.

Another subtlety of prepared statements is that it's quite likely that at least some prepared statements will be prepared and then immediately re-prepared upon execute, due to the statement's original connection being returned to the pool and grabbed by another user during the interval between `db.Prepare()` and `stmt.Exec()`. If you use a network traffic capture inspection tool such as VividCortex, you'll be able to verify this. For example, in one of the VividCortex blog posts we analyzed [single-use prepared statements](#). Careful inspection reveals that the count of `Prepare()` actually exceeds the count of `Exec()` by a small margin, which is expected due to the phenomenon just mentioned.

Another consequence of how the pool handles connections and prepared statements is that you do not need to explicitly handle problems with prepared statements being invalidated by, for example, a failed connection that was killed or timed out server-side. The connection pool will handle this for you transparently. In other words you shouldn't write any logic to re-prepare statements, just like you don't need to write any logic to re-connect to the database. There are up to 10 retries hidden within `database/sql`.

Avoiding Prepared Statement Usage

Sometimes it's better to send plaintext SQL to the server than to prepare statements. Why would you want to do this?

1. The statement has no parameters.
2. The statement won't be reused, so the prepare/execute/close cycle is wasteful tripling of network round-trips and extra latency for the client.
3. The database server doesn't support prepared statements. This is the case for Sphinx and MemSQL, for example, both of which support the MySQL wire protocol but not the prepared statement features of it.

Go's `database/sql` package does allow you to bypass the use of prepared statements, and send the query in a one-shot form as plain text, but the driver needs to support it too. Most drivers the authors

are familiar with do offer this support. The driver simply needs to implement Go's `driver.Execer` and `driver.Queryer` interfaces.

The other crucial part of avoiding prepared statements is under control of you, the programmer. To avoid prepared statements, you should do the following:

- Don't explicitly prepare a statement with `db.Prepare()`, obviously.
- Don't call functions such as `db.Query()` with more than one argument.

The latter requirement may mean that you'll have to build statements yourself by concatenating SQL and quoting values. (You might find `fmt.Sprintf()` with the `%q` placeholder to be useful for this.) If you do this, be careful to avoid SQL injection attack vectors. The best way to do this is by validating your inputs. The authors do this with API parameter-handling frameworks that are wrappers around Go's standard `flag` library, so parameters are strongly typed.

Prepared Statement Parameter Syntax

Parameter syntax is up to the driver and/or the backend database, and `database/sql` doesn't know about it. It is therefore variable depending on which backend database you're using. Here's a quick list of several popular databases and how they handle it:

- **MySQL** uses question-mark parameters `?` that must be matched by an equal number of values during statement execution.
- **PostgreSQL** uses numbered parameters `$1`, `$2` and so on. These can be reused within the statement, so the number of values you pass in during execution might differ from the total number of placeholders. For example, `SELECT $1, $2, $2 FROM mytable` would be executed with only 2 values.
- **Oracle** uses named parameters preceded by colons, such as `:user`.
- **SQLite** accepts both MySQL's and PostgreSQL's syntax.

Working With Transactions

Transactions in `database/sql` are top-level data types like statements and results. And like all the others, you'll need to know how they work to avoid tripping over their subtleties. But before we dig into them, let's discuss the wrong way to work with transactions. The following code will not do the right thing:

```
_, err = db.Exec("BEGIN")
_, err = db.Exec("UPDATE account SET balance = 100 WHERE user = 83")
_, err = db.Exec("COMMIT")
```

Why? Because of the underlying connection pool. There's no guarantee those statements were executed on the same connection. You could have started a transaction (and left it open and idle!) on one connection, updated the `account` table on another connection, and committed some other connection's in-flight transaction. Don't do this!



The database is not in a transaction. The transaction only includes the `sql.Tx` object.

Creating statefulness and binding things to a single connection is exactly what a `sql.Tx` is for, and that's what you should do instead. The essence is as follows:

- You create a `sql.Tx` with `db.Begin()`.
- It removes exactly one connection from the pool and keeps it until it's finished.
- The driver is instructed to start a transaction on that connection.
- The connection, its transaction, and the `tx` variable are coupled, but the `tx` and the `db` are disconnected from each other.
- The lifetime of the transaction and the `tx` ends with `tx.Commit()` or `tx.Rollback()` and the variable is invalid after that.

Let's dig into these and see how transactions work. First, after you use `db.Begin()` to create the `sql.Tx`, **you should operate solely on the `tx` variable, and ignore the `db` for anything that needs to work**

within the transaction. This is because the **db** isn't in a transaction, the **tx** is! This is a common source of confusion for programmers. Code like the following is another buggy anti-pattern:

```
tx, err := db.Begin()
// ...
_, err = db.Exec("UPDATE account SET balance = 100 WHERE user = 83")
// ...
err = tx.Commit()
```

The programmer might not realize it, but the **UPDATE** did not happen within the context of the transaction. Instead of **db.Exec()**, the code should use **tx.Exec()**. Study this if it's confusing, because it's important. On the database server, the transaction is scoped to a single connection; in the code, the connection is bound to the **tx** variable and not available through the **db** anymore. When you call methods on the **db**, you're operating on a different connection, which doesn't participate in the transaction.

The **tx** variable, as we've seen previously with prepared statements, has all the familiar methods with the same signatures: **Query()**, **Exec()**, and so on. There's even a **tx.Prepare()** to prepare statements that are bound solely to the transaction. However, prepared statements work differently within a **tx**. A **sql.Stmt**, when associated with a **sql.Tx**, is bound to the one and only one underlying connection to the database, so there's no auto-repreparing on different connections.

To elucidate this a bit further, a **stmt** that was prepared from a **db** is invalid on the **tx**, and a **stmt** that was prepared from a **tx** is valid only on the **tx**. There is a way to "clone" a **stmt** into the scope of the **tx** by using **tx.Stmt()** but this seems a bit kludgy at the moment, because in the current implementation, it results in the statement being reprepared even if it has already been prepared on the **tx**'s connection.

Within the scope of a **tx**, the usual implicit logic of retrying 10 times also is disabled. You'll need to be prepared to retry statements or

restart the entire transaction yourself, as appropriate for the scenario. This is meat-and-potatoes behavior for transactional databases, naturally. You've always needed to be ready to handle deadlocks and rollbacks when dealing with transactions. And of course a transaction can't be started on one connection and continued on another in most databases.

There's No Concurrency Within A Transaction

There's another thing you should know about working with a `sql.Tx`. Because there's only a single connection, you have to do all of your operations serially, finishing every database interaction before beginning a new one. This is in contrast to what you can do in the usual non-transactional connection-pooled code you might write. For example, the following excerpted code is just fine in normal usage without a transaction:

```
rows, _ := db.Query("SELECT id FROM master_table")
for rows.Next() {
    var mid, did int
    rows.Scan(&mid)
    db.QueryRow("SELECT id FROM detail_table WHERE master = ?", mid).Scan(&did)
}
```

This is okay because when you're working with a `db` variable, the inner statement will remove a new connection from the pool and use it, so the loop will effectively use (at least) two connections. (Inside the loop, the first connection is busy fetching rows for the loop, so it's not available for the `QueryRow()` to use.)

In the scope of a `tx`, however, that won't work:

```
rows, _ := tx.Query("SELECT id FROM master_table")
for rows.Next() {
    var mid, did int
    rows.Scan(&mid)
    tx.QueryRow("SELECT id FROM detail_table WHERE master = ?", mid).Scan(&did) // **BOOM**
}
```

If you do that, you'll be trying to start a new query on the `tx`'s

connection, but it's busy in row-fetching mode and that won't work. Our example is rather silly and could be replaced by a **JOIN**, and doesn't even need to be in a transaction for that matter because it's not modifying any data, but hopefully you see the point. Within the scope of a **tx**, you have to finish each statement before you start the next one.

On the other hand, a corollary also holds. If you need to do some inner-loop fetching within the context of a **tx**, but it doesn't need to participate in the transaction itself for some reason, you could use the **db** to perform it instead. Just be careful that your code doesn't ambush some other programmer later!

There's another, more subtle, way that you can cause concurrent access to connections in a way that is out-of-sync with the backend driver and network protocol. This is by deferring execution of things like **stmt.Close()**, causing them to occur after the **tx** releases its connection back to the pool. If this happens, the deferred call could happen while another operation is using the connection. Here's an example, which omits error handling and detail for brevity:

```
tx, _ := db.Begin()
defer tx.Rollback()
stmt, _ := tx.Prepare("INSERT...")
defer stmt.Close() // will execute when func returns
// Use stmt
tx.Commit() // the connection is returned to the pool
return      // stmt.Close() executes, but conn might already be in-use again
```

To avoid this problem, ensure you **Close()** all transaction-related resources before finishing the transaction. This is actually a [bug](#) in **database/sql** and may be fixed by the time you read this.

Using Transactions For Convenience

Sometimes you want the guarantee that your statements are bound to a single connection, but you don't want to actually create a transaction against the database. Why would you want this? Here are some sample reasons:

1. Connection-specific state, such as temp tables or user-defined

variables, or setting the current database with **USE** or similar.

2. Limiting concurrency and avoiding unwanted connections to the database.
3. Explicit locks.
4. The use of database-specific extensions of behaviors.

If this is necessary, you can take advantage of a **sql.Tx** as a way to access one and only one underlying connection. If you don't want the transaction that comes along with it, you could commit it by calling **tx.Exec("COMMIT")** or similar. We'd normally advise you not to mingle transaction-related SQL commands with **database/sql** function calls related to transactions, but it might work for you. Whether this causes problems is dependent on your driver and backend database. Some databases communicate whether the connection has an active transaction by a signal in the network protocol, and some drivers might respect that and throw an error. The authors remember vaguely a distant past experience with a database and driver (in another language) that would throw an error when trying to finish a transaction that wasn't open anymore. Caveat emptor.

Another possibility would be to create a new **db** variable with **sql.Open()** and set its pool size and max idle connections to 1. Used judiciously, this should have the same effect, but it's not something we'd recommend scattering widely throughout your codebase, nor would we recommend doing this at high speed. Opening and closing database connections rapidly makes some databases (and operating systems' networking stacks) very unhappy.

An alternative, also not recommended particularly highly, would be to use your database driver directly instead of through the **database/sql** interface. That is beyond the scope of this book, so we're just tossing it out there for you to consider.

Error Handling

Idiomatic error handling in Go is to explicitly and immediately check

for an error after every function call that can return an error, and it's no different in `database/sql`. However, for the sake of brevity, we've omitted some error handling in several code listings thus far.

In general, all of the method calls we've seen to this point can return an error, even when we didn't show it. There is one place, however, where you should check for an error that isn't returned by a function call: after the `rows.Next()` loop. We covered the usage of `rows.Err()` previously.

The other special consideration for error-checking is how to inspect and handle errors that the database returns. You might find yourself doing string-matching, for example, to try to catch errors such as a deadlock:

```
if strings.Contains(err.Error(), "Deadlock found") {
    // Handle the error
}
```

But what if the database's language is not set to English, and error codes are returned in Elbonian? Oops. Plus, string-matching like this is just a code smell.

The solution for this depends on the driver and the database. Although some databases return ANSI-standard error codes for some operations, these really are not specific enough for use in most applications. Instead, it's better to deal with the database's own error codes, which are usually very granular and allow you to isolate exactly what happened.

To do this, you'll have to import the driver and use a type assertion to get access to the driver-specific struct underlying the error, like this example using the MySQL driver:

```
if driverErr, ok := err.(*mysql.MySQLError); ok {
    if driverErr.Number == 1213 {
        // Handle the error
    }
}
```



In general, as with all Go code, you should always check and handle all returned errors immediately.

That's better, but still has a code smell. What's that magic number 1213? It's much better to use a defined list of error numbers. Alas, the driver we like doesn't have such a list in it, for various reasons. Fortunately, VividCortex provides one at github.com/VividCortex/mysqlerr. Now the code can be cleaned up more:

```
if driverErr, ok := err.(*mysql.MySQLError); ok {
    if driverErr.Number == mysqlerr.ER_LOCK_DEADLOCK {
        // Handle the error
    }
}
```

Much better. In the most popular PostgreSQL driver, you can use driver-provided types and a driver-provided error list too.

Using Built-In Interfaces

As with all of Go's standard library, `database/sql` uses interfaces heavily to make the magic happen without creating tight coupling between bits of code. And these standard interfaces can be used to great advantage in your own code. There are two interfaces that are involved in passing data into the database and retrieving it back:

- `driver.Valuer` influences how values are transformed as they are sent to the database.
- `sql.Scanner` influences how values are transformed upon retrieval.

You can think of these as filters you can insert into the process of reading and writing from the database.

Why would you want this? As a simple example, suppose you want to ensure that all string values of a certain type are always lowercased when sent to the database, and just in case mixed-case data is present in the database somehow, you also want to lowercase it when reading from the database. We can create a

data type, let's say `LowercaseString`, that enforces these things transparently and keeps the programmer's code clean and simple. This is an overly simplistic example, but the general idea is that we're inserting a `strings.ToLower()` into the process of reading and writing the data. Now, you'd use these types in your code, instead of the `string` type, as parameters when inserting data and as destination variables when scanning. Here's a complete code sample you can examine ([download](#)):

```
package main

import (
    "database/sql"
    "database/sql/driver"
    "errors"
    _ "github.com/go-sql-driver/mysql"
    "log"
    "strings"
)

type LowercaseString string

// Implements driver.Valuer.
func (ls LowercaseString) Value() (driver.Value, error) {
    return driver.Value(strings.ToLower(string(ls))), nil
}

// Implements sql.Scanner. Simplistic -- only handles string and []byte
func (ls *LowercaseString) Scan(src interface{}) error {
    var source string
    switch src.(type) {
    case string:
        source = src.(string)
    case []byte:
        source = string(src.([]byte))
    default:
        return errors.New("Incompatible type for LowercaseString")
    }
    *ls = LowercaseString(strings.ToLower(source))
    return nil
}

func main() {
    db, err := sql.Open("mysql",
        "root:@tcp(:3306)/test")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    _, err = db.Exec(
        "CREATE TABLE IF NOT EXISTS test.hello(world varchar(50))")
    if err != nil {
        log.Fatal(err)
    }
}
```

```

_, err = db.Exec("DELETE FROM test.hello")
if err != nil {
    log.Fatal(err)
}

// Insert a row that's not lowercased, and one that is.
var normalString string = "I AM UPPERCASED NORMAL STRING"
var lcString LowercaseString = "I AM UPPERCASED MAGIC STRING"

_, err = db.Exec("INSERT INTO test.hello VALUES(?), (?)", normalString, lcString)
if err != nil {
    log.Fatal(err)
}

rows, err := db.Query("SELECT * FROM test.hello")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()
for rows.Next() {
    var s1 LowercaseString
    err = rows.Scan(&s1)
    if err != nil {
        log.Print(err)
    }
    log.Print(s1)
}
}

```

If you run this code, it will print out the following:

```

$ go run lowercase.go
2014/12/17 16:08:14 i am uppercased normal string
2014/12/17 16:08:14 i am uppercased magic string

```

As you can see, both rows are apparently lowercased. But if you look in the database, you'll see a different picture:

```

mysql> select * from test.hello;
+-----+
| world                |
+-----+
| I AM UPPERCASED NORMAL STRING |
| i am uppercased magic string |
+-----+

```

This is because when we inserted into the database, we used one normal `string` variable, which got inserted as-is, and one `LowercaseString` variable, which got lowercased on the way into the database. But while reading these rows back, we used a `LowercaseString` as a destination variable, and both of the rows were transformed to lowercase, so the program printed them out in lowercase.



Learning how the `sql.Scanner` and `driver.Valuer` interfaces work will take your usage of database/sql to a completely new level of power and elegance.

This is a very simple example. Real-world examples include much more useful things, such as:

- Enforcing validation of data that must be formatted in a specific way.
- Transforming data into a uniform format.
- Compressing and decompressing data transparently.
- Encrypting and decrypting data transparently.

Here's a sample implementation of gzip compression and decompression, courtesy of Jason Moiron's [blog](#):

```
type GzippedText []byte

func (g GzippedText) Value() (driver.Value, error) {
    b := make([]byte, 0, len(g))
    buf := bytes.NewBuffer(b)
    w := gzip.NewWriter(buf)
    w.Write(g)
    w.Close()
    return buf.Bytes(), nil
}

func (g *GzippedText) Scan(src interface{}) error {
    var source []byte
    // let's support string and []byte
    switch src.(type) {
    case string:
        source = []byte(src.(string))
    case []byte:
        source = src.([]byte)
    default:
        return errors.New("Incompatible type for GzippedText")
    }
    reader, _ := gzip.NewReader(bytes.NewReader(source))
    defer reader.Close()
    b, err := ioutil.ReadAll(reader)
    if err != nil {
        return err
    }
    *g = GzippedText(b)
    return nil
}
```

Now the `GzippedText` type can be used just as easily as a `[]byte` in your source code. At VividCortex, we use similar techniques to transparently encrypt all of our customers' sensitive data when we insert it into our databases. You can read more about that on [our blog post about encryption](#).

Database Drivers

We've alluded several times to third-party database drivers, but what are they, really? And what do they do? It would be great to write a manual for how to create a driver, but that's a little out of scope for this book. Instead we'll cover what they do (briefly) and how they work, and list some good open-source ones you might be interested in.

In brief, the driver's responsibilities are:

1. To open a connection to the database and communicate over it. The driver need not implement any kind of pooling or caching of connections, because `database/sql` does that itself. The connection must support preparing statements, beginning transactions, and closing.
2. To implement `Rows`, an iterator over an executed query's results.
3. To implement an interface for examining an executed statement's results.
4. To implement prepared statements that can be executed and closed.
5. To implement transactions that can be committed or rolled back.
6. To implement bidirectional conversions between values as provided by the database and values in Go.

Drivers can optionally implement a few nice-to-have functionalities as well, most of which signal that the driver and database support a fast-path operation for specific things (such as querying the database directly without using a prepared statement).



Be sure to spend some time learning your preferred driver thoroughly, so you can take advantage of everything it offers.

Drivers become available by registering themselves with `database/sql` via the `sql.Register()` call. They register under the name you'll use in `sql.Open()`. This is done with an `init()` function, similar to the following:

```
func init() {
    sql.Register("mysql", &MySQLDriver{})
}
```

This `init` function executes when the package is imported:

```
import _ "github.com/go-sql-driver/mysql"
```

You can find a list of loaded drivers by calling `sql.Drivers()`, by the way. Here are some of the drivers that seem to be good quality and idiomatic Go code:

- **MySQL:** github.com/go-sql-driver/mysql
- **PostgreSQL:** github.com/lib/pq

You can find more drivers on the [Go wiki page for drivers](#).

If your application is tightly bound to the underlying database (as most are), you'll likely want to get to know your preferred driver well. For example, if you use the PostgreSQL driver we just mentioned, you might be interested in some of the extra bits it exports, such as a `NullBool` type and helpful error-handling functionality. Be sure to read your driver's documentation carefully to learn about all these little goodies.

Common Pitfalls

As you've seen, although the surface area of `database/sql` is pretty small, there's a lot you can do with it. That includes a lot of places you can trip up and make a mistake. This section is dedicated to all the mistakes we've made, in hopes that you won't make them yourself.

Deferring inside a loop. A long-lived function with a query inside a loop, and defer `rows.Close()` inside the loop, will cause both memory and connection usage to grow without bounds.

Opening many db objects. Make a global `sql.DB`, and don't open a new one for, say, every incoming HTTP request your API server should respond to. Otherwise you'll be opening and closing lots of TCP connections to the database. It's a lot of latency, load, and TCP connections in `TIME_WAIT` status.

Not doing `rows.Close()` when done. Forgetting to close the `rows` variable means leaking connections. Combined with growing load on the server, this likely means running into `max_connections` errors or similar. Run `rows.Close()` as soon as you can, even if it'll later be run again (it's harmless). Chain `db.QueryRow()` and `.Scan()` together for the same reason.

Single-use prepared statements. If a prepared statement isn't going to be used more than once, consider whether it makes sense to assemble the SQL with `fmt.Sprintf()` and avoid parameters and prepared statements. This could save two network round-trips, a lot of latency, and potentially wasted work.

Prepared statement bloat. If code will be run at high concurrency, consider whether prepared statements are the right solution, since they are likely to be reprepared multiple times on different connections when connections are busy.

Cluttering the code with `strconv` or casts. Scan into a variable of the type you want, and let `.Scan()` convert behind the scenes for you.

Cluttering the code with error-handling and retry. Let `database/sql` handle connection pooling, reconnecting, and retry logic for you.

Forgetting to check errors after `rows.Next()`. Don't forget that the `rows.Next()` loop can exit abnormally.

Using `db.Query()` for non-SELECT queries. Don't tell Go that you want to iterate over a result set if there won't be one, or you'll leak connections.

Assuming that subsequent statements use the same connection.

Run two statements one after another and they're likely to run on two different connections. Run `LOCK TABLES tb11 WRITE` followed by `SELECT * FROM tb11` and you're likely to block and wait. If you need a guarantee of a single statement being used, you need to use a `sql.Tx`.

Accessing the db while working with a tx. A `sql.Tx` is bound to a transaction, but the `db` is not, so access to it will not participate in the transaction.

Being surprised by a NULL. You can't scan a `NULL` into a variable unless it is one of the `NullXXX` types provided by the `database/sql` package (or one of your own making, or provided by the driver). Examine your schema carefully, because if a column can be `NULL`, someday it *will* be, and what works in testing might blow up in production.

Passing a uint64 as a parameter. For some reason the `Query()`, `QueryRow()`, and `Exec()` methods don't accept parameters of type `uint64` with the most significant bit set. If you start out small and eventually your numbers get big, they could start failing unexpectedly. Convert them to strings with `fmt.Sprintf()` to avoid this.



Conclusion

We hope you've enjoyed this book and that it helps you avoid lost time and other problems. After we learned `database/sql` through much production usage and reading of its source code, we wanted to give that knowledge to the world of Go programmers. If you have any suggestions for improvements, please write to us at info@vividcortex.com.



About VividCortex:

VividCortex is SaaS database performance intelligence for your systems. The database is the heart of most applications, but it's also the part that's hardest to scale, manage, and optimize even as it's growing 50% year over year. VividCortex has developed a suite of unique technologies that significantly eases this pain for the entire IT department. Unlike traditional monitoring, we measure and analyze the system's work and resource consumption. This leads directly to better performance for IT as a whole, at reduced cost and effort.

If you are interested in the benefits of effective database and resource monitoring, [sign up](#) for a free trial today. A look at what is actually happening on your servers will most certainly surprise you. Installation is simple and will take you far less time than reading this eBook.