Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 17

Database Access

Almost every real-world application reads and writes a database. Go's `database/sql` package plays the same role as Java's JDBC: it gives you a standard interface that works with any database driver, so your application code stays independent of the specific database engine underneath it. If you already know JDBC, you will recognize most of the ideas here — the idioms are just different enough to matter.

Coming from Java, you may expect a database layer to mean either JDBC's verbose `try/catch/finally` ceremony or a heavyweight ORM like Hibernate with its sessions, lazy-loading proxies, and dialect configuration. Go takes neither extreme: `database/sql` is a thin, standard-library layer that hands you rows and lets you write SQL directly, with no annotations, no entity manager, and no XML. The result is less magic and less to learn — you write the queries, you `Scan` the results, and connection pooling comes built in rather than bolted on.

The `sql.DB` Connection Pool

In JDBC you open a `Connection` to represent a single database connection, and you typically manage a `DataSource` connection pool separately. Go combines these two concepts: `sql.DB` **is** the connection pool.

```
import (
    "database/sql"
    "log"

    _ "github.com/lib/pq" // register the Postgres driver; see the Driver Registration section
)

func main() {
    db, err := sql.Open("postgres", "postgres://user:pass@localhost/music?sslmode=disable")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()
    // db is ready to use; no actual connection has been made yet
}
```

Wut: `sql.Open` does **not** open a connection. It validates the driver name (and, for most drivers, the DSN format) and returns a pool handle. The first real connection is made lazily on the first query. To verify connectivity at startup, call `db.PingContext(ctx)` immediately after `sql.Open`.

`sql.DB` manages a pool of underlying connections and is **safe for concurrent use by multiple goroutines**. You should create one `sql.DB` per database and share it across your whole application — creating a new `sql.DB` per request is a common mistake that exhausts connection limits quickly.

You can tune the pool with a handful of methods:

```
db.SetMaxOpenConns(25)      // maximum number of open connections
db.SetMaxIdleConns(25)      // maximum number of idle connections kept in the pool
db.SetConnMaxLifetime(time.Hour) // maximum age of a connection before it is recycled
```

The Java JDBC analogy:

JDBC	Go `database/sql`
`DataSource`	`*sql.DB`
`Connection`	internal, managed by the pool
`PreparedStatement`	`*sql.Stmt`
`ResultSet`	`*sql.Rows`
`SQLException`	`error` (idiomatic Go)

Querying the Database

Every query method has a plain form (`Query`, `QueryRow`, `Exec`) and a context-aware form (`QueryContext`, `QueryRowContext`, `ExecContext`). Always use the context-aware forms in production code; they respect cancellation and deadlines (see Chapter 12).

```
// Always prefer these:
db.QueryContext(ctx, query, args...)     // multiple rows
db.QueryRowContext(ctx, query, args...)  // exactly one row
db.ExecContext(ctx, query, args...)      // INSERT / UPDATE / DELETE
```

Trap: The plain `db.Query`, `db.QueryRow`, and `db.Exec` methods use `context.Background()` internally and cannot be cancelled. Using them in a web handler or an RPC server means the query runs to completion even after the client disconnects. Always pass a context.

Querying Multiple Rows

`db.QueryContext` returns `*sql.Rows`, which you iterate with `rows.Next()`.

```
func listSongs(ctx context.Context, db *sql.DB, artist string) ([]Song, error) {
    rows, err := db.QueryContext(ctx,
        "SELECT id, title, artist FROM songs WHERE artist = $1", artist)
    if err != nil {
        return nil, err
    }
    defer rows.Close() // always close rows when done

    var songs []Song
    for rows.Next() {
        var s Song
        if err := rows.Scan(&s.ID, &s.Title, &s.Artist); err != nil {
            return nil, err
        }
        songs = append(songs, s)
    }
```

```
    // rows.Err() reports any error that stopped iteration early
    if err := rows.Err(); err != nil {
        return nil, err
    }
    return songs, nil
}
```

Trap: If you forget `defer rows.Close()` and the function returns early — a `Scan` error, a mid-loop `return` — the connection borrowed from the pool is never returned. (Fully iterating to the end closes the rows automatically, but you should not rely on every path reaching the end.) Under load, the pool is exhausted and new queries block forever. Always `defer rows.Close()` immediately after checking the error from `QueryContext`.

Wut: The placeholder syntax for query parameters is **driver-specific**, not part of `database/sql`. Postgres (`lib/pq`, `pgx`) uses `$1`, `$2`, ...; MySQL and SQLite use `?`; Oracle uses `:name`. The examples in this section use `$1`-style Postgres placeholders; switch to `?` if your driver is MySQL or SQLite, and keep each query consistent with the driver you actually opened.

The Java JDBC equivalent pattern is:

```
try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, artist);
    try (ResultSet rs = ps.executeQuery()) {
        while (rs.next()) { ... }
    }
}
```

Go's `defer rows.Close()` replaces the `try-with-resources` block.

Querying a Single Row

When you expect exactly one row, use `QueryRowContext`. It returns `*sql.Row`, and you call `Scan` on it directly — no loop needed.

```
func getSong(ctx context.Context, db *sql.DB, id int) (Song, error) {
    var s Song
    err := db.QueryRowContext(ctx,
        "SELECT id, title, artist FROM songs WHERE id = $1", id).
        Scan(&s.ID, &s.Title, &s.Artist)
    if err != nil {
        return Song{}, err
    }
    return s, nil
}
```

If no row matches the query, `Scan` returns the sentinel error `sql.ErrNoRows`. Use `errors.Is` to test for it (Chapter 9):

```
if errors.Is(err, sql.ErrNoRows) {
    return Song{}, fmt.Errorf("song %d not found", id)
}
```

Executing Non-Query Statements

`ExecContext` is for statements that do not return rows: `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, and so on. It returns `sql.Result`, which tells you how many rows were affected and the last inserted auto-increment ID.

```go
func addSong(ctx context.Context, db *sql.DB, title, artist string) (int64, error) {
    result, err := db.ExecContext(ctx,
        "INSERT INTO songs (title, artist) VALUES ($1, $2)", title, artist)
    if err != nil {
        return 0, err
    }
    return result.LastInsertId() // not all drivers support this
}
```

Tip: Postgres does not support `LastInsertId`. Use `RETURNING id` in your `INSERT` statement and `QueryRowContext` + `Scan` to read the generated key back.

```go
var id int64
err := db.QueryRowContext(ctx,
    "INSERT INTO songs (title, artist) VALUES ($1, $2) RETURNING id",
    title, artist).Scan(&id)
```

Scanning Results

`rows.Scan` maps database column values into Go variables. You pass a pointer to each destination variable, in the same order as the columns in your `SELECT` list.

```go
var id    int
var title string
var plays int64
rows.Scan(&id, &title, &plays)
```

`Scan` handles type conversions between the database wire type and the Go type. For example, a SQL `INT` column can scan into `int`, `int32`, `int64`, or `string`. If the conversion is not possible, `Scan` returns an error.

Trap: The number of arguments to `Scan` must match the number of columns returned by your query. A mismatch causes a runtime error, not a compile-time error. If you `SELECT *` and the schema changes, `Scan` will fail — prefer explicit column lists.

Nullable Column Values with `sql.Null[T]`

SQL columns can be `NULL`. If you scan a `NULL` value into a plain Go variable such as `string` or `int`, `Scan` returns an error. Before Go 1.22 you used type-specific helpers like `sql.NullString` and `sql.NullInt64`. Go 1.22 introduced the generic `sql.Null[T]`:

```go
type Null[T any] struct {
    V     T    // the value; zero value of T if NULL
    Valid bool // true if the column is not NULL
}
```

Use it when a column may be `NULL`:

```go
type Song struct {
    ID     int
    Title  string
    Artist string
    Album  sql.Null[string] // NULL when the song is a single
    Plays  sql.Null[int64]  // NULL when plays are not tracked
}
```

```go
var s Song
rows.Scan(&s.ID, &s.Title, &s.Artist, &s.Album, &s.Plays)

if s.Album.Valid {
    fmt.Println("Album:", s.Album.V)
} else {
    fmt.Println("Single (no album)")
}
```

Tip: `sql.Null[T]` works with any type `T` accepted by `driver.Value`. For older code (pre-1.22) you will see `sql.NullString`, `sql.NullInt64`, `sql.NullFloat64`, and `sql.NullBool`. They are still valid and work the same way; `sql.Null[T]` just generalizes them.

Transactions

A transaction groups multiple statements into an atomic unit: either all succeed or all are rolled back. In JDBC you call `conn.setAutoCommit(false)` and then `conn.commit()` or `conn.rollback()`. In Go you call `db.BeginTx` to get a `*sql.Tx`, then use `tx.Commit()` or `tx.Rollback()`.

`*sql.Tx` has the same query methods as `*sql.DB`: `QueryContext`, `QueryRowContext`, `ExecContext`, and `PrepareContext`. All statements executed on a `*sql.Tx` run inside the same database transaction.

The Deferred Rollback Pattern

The idiomatic Go pattern for transactions uses `defer` to guarantee a rollback if anything goes wrong:

```go
func transferPlay(ctx context.Context, db *sql.DB, fromID, toID int, count int64) error {
    tx, err := db.BeginTx(ctx, nil) // nil uses the default isolation level
    if err != nil {
        return err
    }
    defer tx.Rollback() // no-op if tx.Commit() has already been called

    _, err = tx.ExecContext(ctx,
        "UPDATE songs SET plays = plays - $1 WHERE id = $2", count, fromID)
    if err != nil {
        return err // defer fires tx.Rollback()
    }

    _, err = tx.ExecContext(ctx,
        "UPDATE songs SET plays = plays + $1 WHERE id = $2", count, toID)
    if err != nil {
        return err // defer fires tx.Rollback()
    }

    return tx.Commit() // defer fires tx.Rollback(), but Rollback after Commit is a no-op
}
```

The key insight: `tx.Rollback()` is a no-op if the transaction has already been committed. So you can `defer tx.Rollback()` unconditionally right after `BeginTx`, and then call `tx.Commit()` at the end of the happy path. If anything returns early with an error, the deferred rollback fires and cleans up.

Tip: This pattern is the Go equivalent of Java's `try { ... conn.commit(); } catch (Exception e) { conn.rollback(); }` boilerplate. It is shorter and harder to forget because `defer` always runs, even on panics.

Trap: After `tx.Commit()` or `tx.Rollback()` is called, the `*sql.Tx` is no longer usable. Any further operations on it return `sql.ErrTxDone`.

Prepared Statements

A prepared statement is a pre-compiled SQL template that can be executed multiple times with different parameter values. In JDBC, `PreparedStatement` is the standard way to parameterize queries. In Go, use `db.PrepareContext` to get a `*sql.Stmt`:

```
func bulkInsert(ctx context.Context, db *sql.DB, songs []Song) error {
    stmt, err := db.PrepareContext(ctx,
        "INSERT INTO songs (title, artist) VALUES ($1, $2)")
    if err != nil {
        return err
    }
    defer stmt.Close()

    for _, s := range songs {
        if _, err := stmt.ExecContext(ctx, s.Title, s.Artist); err != nil {
            return err
        }
    }
    return nil
}
```

`*sql.Stmt` has the same execution methods as `*sql.DB`: `QueryContext`, `QueryRowContext`, and `ExecContext`.

Tip: Prepare the statement once and execute it many times. Preparing a statement on every iteration of a loop defeats the purpose — the database has to parse and plan the query each time anyway.

Wut: Go's `database/sql` transparently re-prepares statements when a connection from the pool is replaced. The `*sql.Stmt` handle is associated with the pool, not a single connection. Under the hood, Go may prepare the same statement on multiple connections as needed. This is different from JDBC, where a `PreparedStatement` is tied to a single `Connection`.

You can also prepare a statement inside a transaction:

```
tx, err := db.BeginTx(ctx, nil)
if err != nil { ... }
defer tx.Rollback()

stmt, err := tx.PrepareContext(ctx, "INSERT INTO songs (title, artist) VALUES ($1, $2)")
if err != nil { ... }
defer stmt.Close()
```

Statements prepared on a `*sql.Tx` are scoped to that transaction.

Driver Registration

`database/sql` is driver-neutral. It defines the API; the actual wire protocol to a specific database is provided by a third-party driver package. Drivers register themselves with `database/sql` in their `init()` function (see Chapter 5). You import the driver package for its side effects — you never call it directly.

```
import _ "github.com/lib/pq"              // PostgreSQL driver
import _ "github.com/mattn/go-sqlite3"    // SQLite driver
import _ "github.com/go-sql-driver/mysql" // MySQL driver
```

The `_` alias discards the package name so the compiler does not complain about an unused import. The `init()` function inside the driver package calls `sql.Register("postgres", &Driver{})` (or equivalent), making the driver available to `sql.Open`.

This is the same blank import pattern introduced in Chapter 1 (for side-effect-only imports). The driver package has a valuable side effect — driver registration — but no exported API that your code calls directly.

Tip: `github.com/mattn/go-sqlite3` wraps the C SQLite library, so it requires cgo — you need `CGO_ENABLED=1` and a working C toolchain to build it. If you want to avoid cgo entirely, use the pure-Go driver `modernc.org/sqlite` (registered under the name `"sqlite"`), which builds anywhere Go does.

Tip: The driver name string you pass to `sql.Open` (e.g., `"postgres"`, `"sqlite3"`, `"mysql"`) must match the name the driver registers in its `init()`. Check the driver's documentation if `sql.Open` returns `unknown driver` — you may be using the wrong name or missing the blank import.

`pgx` — The PostgreSQL Driver

For PostgreSQL specifically, `github.com/jackc/pgx/v5` is the dominant driver and most new Go projects use it directly rather than through `database/sql`. `pgx` exposes the full Postgres wire protocol — named parameters, `COPY`, `LISTEN/NOTIFY`, batch queries — none of which are available through the `database/sql` interface.

You can use `pgx` in two modes:

Mode 1: as a `database/sql` driver (drop-in, no API change):

```
import (
    "database/sql"
    _ "github.com/jackc/pgx/v5/stdlib" // registers "pgx" driver name
)

db, err := sql.Open("pgx", os.Getenv("DATABASE_URL"))
```

Mode 2: native `pgx` API (recommended for new Postgres projects):

```
import "github.com/jackc/pgx/v5/pgxpool"

pool, err := pgxpool.New(ctx, os.Getenv("DATABASE_URL"))
```

`pgxpool.Pool` is a connection pool with the same context-aware query methods as `database/sql` but with access to Postgres-specific features. `pgx` also provides `pgx.CollectRows` and `pgx.RowToStructByName` helpers that scan results directly into structs without manual `Scan` calls.

Tip: Prefer the native `pgx` API for new PostgreSQL projects. Use the `stdlib` wrapper only when you need to share code with a `database/sql`-based library (e.g., `sqlx`, `goose`).

Try It: A Small Music Store

Type this in and run it. It is a complete, self-contained program that opens an in-memory SQLite database, creates a table, inserts a few songs, and queries them back, demonstrating the core APIs from this chapter in one place. SQLite uses `?` placeholders, so every query below matches the driver it opens.

```
package main

import (
    "context"
    "database/sql"
    "fmt"
    "log"

    _ "github.com/mattn/go-sqlite3" // register the SQLite driver
)

type Song struct {
    ID     int
    Title  string
    Artist string
    Album  sql.Null[string]
}

func main() {
    db, err := sql.Open("sqlite3", ":memory:")
    if err != nil {
        log.Fatal(err)
    }
    defer db.Close()

    ctx := context.Background()

    if err := db.PingContext(ctx); err != nil { // verify connectivity
        log.Fatal(err)
    }

    _, err = db.ExecContext(ctx, `
        CREATE TABLE songs (
            id     INTEGER PRIMARY KEY AUTOINCREMENT,
            title  TEXT    NOT NULL,
            artist TEXT    NOT NULL,
            album  TEXT
        )`)
    if err != nil {
        log.Fatal(err)
    }

    // bulk insert using a prepared statement
```

```
stmt, err := db.PrepareContext(ctx,
    "INSERT INTO songs (title, artist, album) VALUES (?, ?, ?)")
if err != nil {
    log.Fatal(err)
}
defer stmt.Close()

inserts := []Song{
    {
        Title:  "Sounds of Slashdot",
        Artist: "San Mehat",
        Album:  sql.Null[string]{V: "DJ Essentials: Trance", Valid: true},
    },
    {
        Title:  "Gamemaster",
        Artist: "Matt Darey & Lost Tribe",
        Album:  sql.Null[string]{V: "DJ Essentials: Trance", Valid: true},
    },
    {
        Title:  "J'ai pas vingt ans !",
        Artist: "Alizée",
        Album:  sql.Null[string]{V: "Mes Courants Électriques...", Valid: true},
    },
    {
        Title:  "Gouryella",
        Artist: "Gouryella",
        Album:  sql.Null[string]{}, // NULL album
    },
}
for _, s := range inserts {
    if _, err := stmt.ExecContext(ctx, s.Title, s.Artist, s.Album); err != nil {
        log.Fatal(err)
    }
}

// query all songs for San Mehat and Matt Darey & Lost Tribe (DJ Essentials: Trance)
rows, err := db.QueryContext(ctx,
    "SELECT id, title, artist, album FROM songs WHERE album = ?", "DJ Essentials: Trance")
if err != nil {
    log.Fatal(err)
}
defer rows.Close()

for rows.Next() {
    var s Song
    if err := rows.Scan(&s.ID, &s.Title, &s.Artist, &s.Album); err != nil {
        log.Fatal(err)
    }
    album := "single"
    if s.Album.Valid {
        album = s.Album.V
    }
    fmt.Printf("%d: %s --- %s (%s)\n", s.ID, s.Title, s.Artist, album)
}
```

```
    if err := rows.Err(); err != nil {
        log.Fatal(err)
    }
}
```

Output:

```
1: Sounds of Slashdot --- San Mehat (DJ Essentials: Trance)
2: Gamemaster --- Matt Darey & Lost Tribe (DJ Essentials: Trance)
```

Once it runs, try these modifications:

- Wrap the four inserts in a transaction using `db.BeginTx` and the deferred rollback pattern, then commit at the end.
- Change the `WHERE album = ?` query to select every song and print `single` for the rows where `s.Album.Valid` is `false`.
- Delete the `db.PingContext` call and the table-creation step, then observe the exact error `QueryContext` returns when the `songs` table does not exist.

Key Points

- `sql.DB` is a **connection pool**, not a single connection; create one instance per database and share it across your application.
- Always use the context-aware methods — `QueryContext`, `QueryRowContext`, `ExecContext` — so that queries respect cancellation and timeouts (Chapter 12).
- `rows.Scan` maps column values into Go variables by position; always `defer rows.Close()` and check `rows.Err()` after the loop. [*no-discard-error*]
- `sql.ErrNoRows` is the sentinel error from `QueryRowContext` when no row matches; test with `errors.Is`.
- `sql.Null[T]` (Go 1.22) wraps nullable column values; `Valid` is `true` when the column is not `NULL`.
- `sql.Tx` groups statements into a transaction; the **deferred rollback pattern** guarantees cleanup on any error path.
- `db.PrepareContext` returns a `*sql.Stmt`; prepare once, execute many times.
- Driver packages (e.g., `github.com/lib/pq`) register themselves via `init()` and must be blank-imported to take effect (Chapter 1).
- For PostgreSQL, prefer `pgx/v5` (`github.com/jackc/pgx/v5`) over `lib/pq`; use the native `pgxpool.Pool` API for full Postgres feature access, or `pgx/v5/stdlib` for a `database/sql`-compatible drop-in.

Exercises

1. **Think about it:** JDBC requires explicit transaction management and connection pooling through a `DataSource`, usually provided by an application server or a library like HikariCP. Go's `database/sql` builds connection pooling directly into `sql.DB`. What are the tradeoffs of each approach? In what situations might you still want an external connection pool in a Go application?

2. **What does this print?**

```
package main

import (
    "database/sql"
    "fmt"
)

func main() {
    a := sql.Null[string]{V: "J'ai pas vingt ans !", Valid: true}
```

```
    b := sql.Null[string]{V: "Gouryella", Valid: false}
    c := sql.Null[int64]{V: 0, Valid: false}

    fmt.Println(a.Valid, a.V)
    fmt.Println(b.Valid, b.V)
    fmt.Println(c.Valid, c.V)
}
```

3. **Calculation:** Trace the following transaction sequence and state whether the database ends up with the row inserted or not, and why.

```
tx, _ := db.BeginTx(ctx, nil)
defer tx.Rollback()

_, err := tx.ExecContext(ctx, "INSERT INTO songs (title, artist) VALUES (?, ?)",
    "Sounds of Slashdot", "San Mehat")
if err != nil {
    return err
}

return tx.Commit()
```

   (Error handling on `BeginTx` is elided to keep the trace short.)

   Case A: `ExecContext` succeeds and `Commit` succeeds. Case B: `ExecContext` succeeds but `Commit` returns an error. Case C: `ExecContext` returns an error.

4. **Where is the bug?**

```
func getArtistSongs(ctx context.Context, db *sql.DB, artist string) ([]string, error) {
    rows, err := db.QueryContext(ctx,
        "SELECT title FROM songs WHERE artist = ?", artist)
    if err != nil {
        return nil, err
    }

    var titles []string
    for rows.Next() {
        var title string
        if err := rows.Scan(&title); err != nil {
            return nil, err
        }
        titles = append(titles, title)
    }
    return titles, nil
}
```

5. **Write a program:** Using `database/sql` and the `github.com/mattn/go-sqlite3` driver, write a program that:

   - Opens an in-memory SQLite database.
   - Creates a `playlists` table with columns `id` (integer primary key autoincrement), `name` (text, not null), and `owner` (text, not null).
   - Inserts at least three rows inside a single transaction using the deferred rollback pattern.
   - Queries and prints all rows using `QueryContext` and `Scan`.
   - Uses `sql.Null[string]` for at least one nullable column (add a `description` column that is nullable).