Using Neo4j with Go

Learn how to interact with Neo4j from Go using the Neo4j Go Driver

Installing the driver

Introduction

In the **Cypher Fundamentals** course, you learned how to query Neo4j using Cypher.

To run Cypher statements in a Go application, you'll need the **Neo4j Go Driver**. The driver acts as a bridge between your Go code and Neo4j, handling connections to the database and the execution of Cypher queries.

Setting up a Go Project

First, create a new Go project and initialize it as a module:

shell:

```
mkdir hello-neo4j

cd hello-neo4j

go mod init graphacademy/hello
```

Installing the Driver

To install the driver, use the `go get` command:

shell:

```
go get github.com/neo4j/neo4j-go-driver/v5
```

Creating a Driver Instance

go:

```go
package main

import (
    "context"
    "fmt"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

func main() {
    driver, err := neo4j.NewDriverWithContext(
        "neo4j://localhost:7687", // (1)
        neo4j.BasicAuth("neo4j", "your-password", ""), // (2)
    )
    if err != nil { panic(err) }
    defer driver.Close(context.Background()) // (3)
}
```

You start by importing the driver and creating an instance using the `neo4j.NewDriverWithContext()` function:

1. The connection string for your Neo4j database
2. Your Neo4j username and password
3. Always close the driver when done using the `defer` statement

> **Best Practice**
>
> Create **one** Driver instance and share it across your entire application.

Verifying Connectivity

You can verify the connection is correct by calling the `VerifyConnectivity()` method.

go:

```go
ctx := context.Background()
err = driver.VerifyConnectivity(ctx)
if err != nil {
    panic(err)
}
```

Verify Connectivity

The `VerifyConnectivity()` method will **return an error** if the connection cannot be made.

Running Your First Query

The `ExecuteQuery()` method executes a Cypher query and returns the results.

go:

```go
ctx := context.Background()
result, err := neo4j.ExecuteQuery(ctx, driver, // (1)
    "RETURN COUNT {()} AS count",
    nil, // (2)
    neo4j.EagerResultTransformer, // (3)
)
if err != nil { panic(err) }

// Get the first record
first := result.Records[0] // (4)
// Print the count entry
count, _ := first.Get("count") // (5)
fmt.Println(count)
```

What is happening here?

1. `neo4j.ExecuteQuery()` runs a Cypher query to get the count of all nodes in the database
2. `nil` means no parameters are passed to the query
3. `EagerResultTransformer` loads all results into memory
4. `Records` contains a slice of the rows returned
5. Values from the `RETURN` clause are accessed using the `Get()` method

Full driver lifecycle

It is good practice to close the driver when you are finished with it. You can use the `defer` statement to automatically close the driver when the function exits.

You can also explicitly call `Close()` to release any resources held by the driver.

go:

```
driver, err := neo4j.NewDriverWithContext(
    "neo4j://localhost:7687",
    neo4j.BasicAuth("neo4j", "your-password", ""),
)
if err != nil {
    panic(err)
}
defer driver.Close(context.Background())
```

go:

```
ctx := context.Background()
driver.Close(ctx)
```

Executing Cypher statements

Introduction

You can use the `neo4j.ExecuteQuery()` function to run one-off Cypher statements or statements that return a small number of records. This function fetches a list of records and loads them into memory.

go:

```
cypher := `
MATCH (p:Person {name: $name})-[r:ACTED_IN]->(m:Movie)
RETURN m.title AS title, r.role AS role
`

name := "Tom Hanks"


result, err := neo4j.ExecuteQuery(ctx, driver, // (1)
    cypher,    // (2)
    map[string]any{"name": name}, // (3)
    neo4j.EagerResultTransformer, // (4)
)
```

1. The function returns a result object and an error.
2. The function expects a Cypher statement as a string as the first argument.
3. Parameters are passed as a map with string keys and any values.
4. The result transformer determines how results are processed.

Using Parameters

It is good practice to use parameters in your queries to avoid malicious code being injected into your Cypher statement.

Handling the Result

The `neo4j.ExecuteQuery()` function returns a `ResultWithContext` object containing:

1. A list of `Record` objects
2. Summary information about the query execution
3. Keys specified in the `RETURN` clause

go:

```
fmt.Println(result.Records) // [...]
fmt.Println(result.Keys)  // [title role]
fmt.Println(result.Summary)  // A summary of the query execution
```

Specifying a database

You can specify a database to query using the `ExecuteQueryWithDatabase()` method.

go:

```go
result, err := neo4j.ExecuteQuery(ctx, driver,
    cypher,
    map[string]any{"name": name},
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithDatabase("recommendations") // Query the 'recommendations' database
)
```

Accessing results

Each row returned by the query is a `Record` object. The `Record` object provides access to the data returned by the query.

You can access any item in the `RETURN` clause using the `Get()` method.

go:

```go
// RETURN a.name AS name, m.title AS title, r.role AS role


for _, record := range result.Records {
    name, _ := record.Get("name")  // Tom Hanks
    title, _ := record.Get("title")  // Toy Story
    role, _ := record.Get("role")    // "Woody"
    fmt.Printf("%s played %s in %s\n", name, role, title) // Tom Hanks played Woody in Toy Story
}
```

Transforming results

The `ExecuteQuery()` method accepts a result transformer as a third argument that allows you to transform the result into an alternative format.

go:

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    cypher,
    map[string]any{"name": name},
    func(result neo4j.ResultWithContext) (any, error) {
        var movies []string
        for _, record := range result.Records {
            title, _ := record.Get("title")
            role, _ := record.Get("role")
            movies = append(movies, fmt.Sprintf("Tom Hanks played %s in %s", role, title))
        }
        return movies, nil
    },
)
```

Transforming results

Rather than returning the standard result, the method will return the output of the transformer function.

go:

```
fmt.Println(result)  // ["Tom Hanks played Woody in Toy Story", ...]
```

Reading and writing

By default, `neo4j.ExecuteQuery()` runs in **WRITE** mode. In a clustered environment, this sends all queries to the cluster leader, putting unnecessary load on the leader.

When you're only reading data, you can optimize performance by setting the routing control to READ mode. This distributes your read queries across all cluster members.

go:

```
result, err := neo4j.ExecuteQuery(ctx, driver,
    cypher,
    map[string]any{"name": name},
    neo4j.EagerResultTransformer,
    neo4j.ExecuteQueryWithReadersRouting(), // (1)
)
```

You can also use `neo4j.ExecuteQueryWithWritersRouting` to explicitly invoke write mode.

Graph types

Introduction

Let’s take a look at the types of data returned by a Cypher query.

The majority of the types returned by a Cypher query are mapped directly to Go types, but some more complex types need special handling.

- Graph types - Nodes, Relationships and Paths
- Temporal types - Dates and times
- Spatial types - Points and distances

> **Returning graph types**
>
> When graph types are returned by a query executed in the Query window, they are visualized in a graph layout.

Table 1. Direct mapping

Go Type	Neo4j Cypher Type
`nil`	`null`
`bool`	`Boolean`
`int64`	`Integer`
`float64`	`Float`
`string`	`String`
`[]byte`	`Bytes [1]`
`[]any`	`List`
`map[string]any`	`Map`

Graph types

The following code snippet finds all movies with the specified title and returns `person`, `acted_in` and `movie`.

go: Return Nodes and Relationships

```
result, err := neo4j.ExecuteQuery(ctx, driver, `
MATCH path = (person:Person)-[actedIn:ACTED_IN]->(movie:Movie {title: $title})
RETURN path, person, actedIn, movie
`, map[string]any{"title": movie}, neo4j.EagerResultTransformer)
```

Nodes

Nodes are returned as a `neo4j.Node` object.

go:

```go
for _, record := range result.Records {
    node, _ := record.Get("movie")
    movieNode := node.(neo4j.Node)
}
```

go: Working with Node Objects

```go
fmt.Println(movieNode.ElementId)            // (1)
fmt.Println(movieNode.Labels)               // (2)
fmt.Println(movieNode.Props)                // (3)

if name, ok := movieNode.Props["name"]; ok { // (4)
    fmt.Println("name:", name)
}
```

1. The `ElementId` property provides access to the node's element ID
   eg. `4:97b72e9c-ae4d-427c-96ff-8858ecf16f88:0`
2. The `Labels` property is a slice containing an array of labels attributed to the Node
   eg. `["Person", "Actor"]`
3. The `Props` property provides access to the node's properties as a map.
   eg. `{name: "Tom Hanks", tmdbId: "31"}`
4. A single property can be retrieved using the `Props` map.

Relationships

Relationships are returned as a `neo4j.Relationship` object.

go:

```go
actedIn, _ := record.Get("actedIn")
relationship := actedIn.(neo4j.Relationship)

fmt.Println(relationship.ElementId)             // (1)
fmt.Println(relationship.Type)                  // (2)
fmt.Println(relationship.Props)                 // (3)

if role, ok := relationship.Props["role"]; ok { // (4)
    fmt.Println("role:", role)
}
```

1. `ElementId` - Internal ID of the relationship (eg. `9876`)
2. `Type` - Type of relationship (eg. `ACTED_IN`)
3. `Props` - Returns relationship properties as a map (eg. `{role: "Woody"}`)
4. Individual properties can be accessed using the `Props` map.

Paths

A path is a sequence of nodes and relationships and is returned as a `neo4j.Path` object.

go:

```go
path, _ := record.Get("path")
pathObj := path.(neo4j.Path)

fmt.Println(pathObj.Start)  // (1)
fmt.Println(pathObj.End)    // (2)
fmt.Println(len(pathObj.Relationships))  // (3)
fmt.Println(pathObj.Relationships)  // (4)
```

1. `Start` - `neo4j.Node` object at the start of the path
2. `End` - `neo4j.Node` object at the end of the path
3. `len(pathObj.Relationships)` - The number of relationships within the path
4. `Relationships` - A slice of `neo4j.Relationship` objects within the path.

Paths are iterable

Use `for _, rel := range pathObj.Relationships` to iterate over the relationships in a path.

Dates and times

Temporal types

The `neo4j` package provides types for working with dates and times in Go.

Temporal types in Neo4j are a combination of date, time and timezone elements.

Table 1. Temporal Types

Type	Description	Date?	Time?	Timezone?
`neo4j.Date`	A tuple of Year, Month and Day	Y		
`neo4j.OffsetTime`	The time of the day with a UTC offset		Y	Y
`neo4j.LocalTime`	A time without a timezone		Y	
`time.Time`	A combination of Date and Time	Y	Y	Y
`neo4j.LocalDateTime`	A combination of Date and Time without a timezone	Y	Y	
`neo4j.Duration`	A period of time			

Writing temporal types

go:

```
res, err := neo4j.ExecuteQuery(ctx, driver, `
CREATE (e:Event {
  startsAt: $datetime,              // (1)
  createdAt: datetime($dtstring),   // (2)
  updatedAt: datetime()             // (3)
})
`,
    map[string]any{
        "datetime": time.Date(2024, 5, 15, 14, 30, 0, 0, time.LoadLocation(
        "dtstring": "2024-05-15T14:30:00+02:00",
    },
    neo4j.EagerResultTransformer,
)
```

When you write temporal types to the database, you can pass Go time objects as parameters to the query or cast the value within a Cypher statement.

This example demonstrates how to:

1. Use a `time.Time` object as a parameter to the query (`<4>`)
2. Cast an `datetime` with an **IANA timezone identifier** within the Cypher statement
3. Get the current date and time using the `datetime()` function.

Reading temporal types

When reading temporal types from the database, you will receive an instance of the corresponding Go type unless you cast the value within your query.

go:

```go
// Query returning temporal types
result, err := neo4j.ExecuteQuery(ctx, driver, `
RETURN date() as date, time() as time, datetime() as datetime, toString(datetime()) as asString
`, nil, neo4j.EagerResultTransformer)

// Access the first record
for _, record := range result.Records {
    // Automatic conversion to Go driver types
    date, _ := record.Get("date")           // neo4j.Date
    time, _ := record.Get("time")           // neo4j.OffsetTime
    datetime, _ := record.Get("datetime")   // time.Time
    asString, _ := record.Get("asString")   // string
}
```

Working with Durations

go:

```
startsAt := time.Now()
eventLength := neo4j.DurationOf(time.Hour + 30*time.Minute)
endsAt := startsAt.Add(time.Duration(eventLength))


neo4j.ExecuteQuery(ctx, driver, `
CREATE (e:Event {
  startsAt: $startsAt, endsAt: $endsAt,
  duration: $eventLength, // (1)
  interval: duration('P30M') // (2)
})`,
    map[string]any{
        "startsAt": startsAt,
        "endsAt": endsAt,
        "eventLength": eventLength,
    },
    neo4j.EagerResultTransformer,
)
```

Durations represent a period of time and can be used for date arithmetic in both Go and Cypher. These types can also be created in Go or cast within a Cypher statement.

1. Pass an instance of `neo4j.Duration` to the query
2. Use the `duration()` function to create a `Duration` object from an ISO 8601 format string

Calculating durations

You can use Go's `time` package methods to calculate the duration between two date or time objects.

Spatial types

Points and locations

Neo4j has built-in support for two-dimensional and three-dimensional spatial data types. These are referred to as **points**.

A point may represent geographic coordinates (longitude, latitude) or Cartesian coordinates (x, y).

In Go, points are represented by the `neo4j.Point2D` and `neo4j.Point3D` types, which provide methods to access the coordinates and SRID of the point.

> **SRID**
>
> The **Spatial Reference Identifier** (SRID) is a unique identifier used to define the type of coordinate system used.

Cypher Type	Go Type	SRID	3D SRID
Point (Cartesian)	`neo4j.Point2D` / `neo4j.Point3D`	`7203`	`9157`
Point (WGS-84)	`neo4j.Point2D` / `neo4j.Point3D`	`4326`	`4979`

CartesianPoint

A Cartesian Point defines a point with x and y coordinates. An additional z value can be provided to define a three-dimensional point.

You can create a cartesian point by creating a `neo4j.Point2D` or `neo4j.Point3D` struct, or by passing `x`, `y` and optionally `z` values to the point function in Cypher.

go:

```go
import "github.com/neo4j/neo4j-go-driver/v5/neo4j"


// 2D Cartesian point
point2D := neo4j.Point2D{
    X: 1.23,
    Y: 4.56,
    SpatialRefId: 7203, // Cartesian SRID
}


// 3D Cartesian point
point3D := neo4j.Point3D{
    X: 1.23,
    Y: 4.56,
    Z: 7.89,
    SpatialRefId: 9157, // 3D Cartesian SRID
}
```

CartesianPoint

The driver will convert `point` data types created with x, y and z values to instances of the `neo4j.Point2D` or `neo4j.Point3D` types.

The values can be accessed using the `X`, `Y`, `Z` and `SpatialRefId` fields.

go:

```go
result, err := neo4j.ExecuteQuery(ctx, driver, `
RETURN point({x: 1.23, y: 4.56, z: 7.89}) AS threeD
`, nil, neo4j.EagerResultTransformer)


if err == nil && len(result.Records) > 0 {
    point, _ := result.Records[0].Get("threeD")
    if p3d, ok := point.(neo4j.Point3D); ok {
        fmt.Printf("X: %f, Y: %f, Z: %f, SRID: %d\n",
            p3d.X, p3d.Y, p3d.Z, p3d.SpatialRefId)
    }
}
```

WGS84Point

A WGS *(World Geodetic System)* point consists of longitude (`X`) and latitude (`Y`) values. An additional height (`Z`) value can be provided to define a three-dimensional point.

You can create a WGS84 point by creating a `neo4j.Point2D` or `neo4j.Point3D` struct with the appropriate SRID, or by passing `longitude` , `latitude` and `height` values to the point function in Cypher.

go:

```go
// London coordinates (2D)
london := neo4j.Point2D{
    X: -0.118092,  // longitude
    Y: 51.509865,  // latitude
    SpatialRefId: 4326, // WGS-84 SRID
}


// The Shard with height (3D)
shard := neo4j.Point3D{
    X: -0.086500,  // longitude
    Y: 51.504501,  // latitude
    Z: 310,        // height in meters
    SpatialRefId: 4979, // WGS-84 3D SRID
}
```

WGS84Point

The driver will return `neo4j.Point2D` or `neo4j.Point3D` objects when `point` data types are created with `latitude` and `longitude` values in Cypher.

go:

```go
result, err := neo4j.ExecuteQuery(ctx, driver, `
RETURN point({ latitude: 51.5, longitude: -0.118, height: 100 }) AS point
`, nil, neo4j.EagerResultTransformer)


if err == nil && len(result.Records) > 0 {
    point, _ := result.Records[0].Get("point")
    if p3d, ok := point.(neo4j.Point3D); ok {
        longitude := p3d.X
        latitude := p3d.Y
        height := p3d.Z
        srid := p3d.SpatialRefId
        fmt.Printf("Lon: %f, Lat: %f, Height: %f, SRID: %d\n",
            longitude, latitude, height, srid)
    }
}
```

Distance

The `point.distance` function can be used to calculate the distance between two points with the same SRID.

The result is a `float64` representing the distance in a straight line between the two points.

SRIDs must be compatible

If the SRID values are different, the function will return an error.

go:

```go
// Create two Cartesian points
point1 := neo4j.Point2D{X: 1.23, Y: 4.56, SpatialRefId: 7203}
point2 := neo4j.Point2D{X: 2.34, Y: 5.67, SpatialRefId: 7203}

// Query the distance using Cypher
result, err := neo4j.ExecuteQuery(ctx, driver, `
RETURN point.distance($p1, $p2) AS distance
`, map[string]any{
    "p1": point1,
    "p2": point2,
}, neo4j.EagerResultTransformer)

if err == nil && len(result.Records) > 0 {
    distance, _ := result.Records[0].Get("distance")
    fmt.Printf("Distance: %f\n", distance.(float64))
}
```

Transaction management

Introduction

In the previous module, you learned how to execute one-off Cypher statements using the `ExecuteQuery()` method.

The drawback of this method is that the entire record set is only available once the final result is returned. For longer running queries or larger datasets, this can consume a lot of memory and a long wait for the final result.

In a production application, you may also need finer control of database transactions or to run multiple related queries as part of a single transaction.

Transaction functions allow you to run multiple queries in a single transaction while accessing results immediately.

> **Understanding Transactions**
>
> Neo4j is an ACID-compliant transactional database, which means queries are executed as part of a single atomic transaction. This ensures your data operations are consistent and reliable.

Sessions

To execute transactions, you need to open a session. The session object manages the underlying database connections and provides methods for executing transactions.

go:

```
session := driver.NewSession(ctx, neo4j.SessionConfig{})
defer session.Close(ctx)
// Call transaction functions here
```

Using `defer session.Close(ctx)` will automatically close the session and release any underlying connections when the function exits.

Specifying a database

In a multi-database instance, you can specify the database to use when creating a session using the `Database` field in `SessionConfig`.

Transaction functions

The session object provides two methods for managing transactions:

- `Session.ExecuteRead()`
- `Session.ExecuteWrite()`

If the entire function runs successfully, the transaction is committed automatically. If any errors occur, the entire transaction is rolled back.

> **Transient errors**
>
> These functions will also retry if the transaction fails due to a transient error, for example, a network issue.

Unit of work patterns

go:

```go
func createPerson(tx neo4j.ManagedTransaction, name string, age int64) (nec
    result, err := tx.Run(ctx, `
    CREATE (p:Person {name: $name, age: $age})
    RETURN p
    `, map[string]any{"name": name, "age": age}) // (2)


    record, err := result.Single(ctx)


    node, _ := record.Get("p")
    return node.(neo4j.Node), nil
}
```

1. The first argument to the transaction function is always a `ManagedTransaction` object. Any additional arguments are passed from the call to `Session.ExecuteRead` / `Session.ExecuteWrite`.

2. The `Run()` method on the `ManagedTransaction` object is called to execute a Cypher statement.

Multiple Queries in One Transaction

You can execute multiple queries within the same transaction function to ensure that all operations are completed or fail as a single unit.

go:

```go
func transferFunds(tx neo4j.ManagedTransaction, fromAccount, toAccount stri
    // Deduct from first account
    _, err := tx.Run(ctx,
        "MATCH (a:Account {id: $from}) SET a.balance = a.balance - $amount"
        map[string]any{"from": fromAccount, "amount": amount},
    )
    if err != nil { return err }

    // Add to second account
    _, err = tx.Run(ctx,
        "MATCH (a:Account {id: $to}) SET a.balance = a.balance + $amount",
        map[string]any{"to": toAccount, "amount": amount},
    )
    return err
}
```

Transaction state

Transaction state

Transaction state is maintained in the DBMS's memory, so be mindful of running too many operations in a single transaction. Break up very large operations into smaller transactions when possible.

Handling outputs

The `ManagedTransaction.Run()` method returns a `Result` object. The records contained within the result can be accessed as soon as they are available.

The result must be consumed within the transaction function.

The `Consume()` method discards any remaining records and returns a `Summary` object that can be used to access metadata about the Cypher statement.

The `Session.ExecuteRead` / `Session.ExecuteWrite` function will return the result of the transaction function upon successful execution.

go: Consuming results

```
session := driver.NewSession(ctx, neo4j.SessionConfig{})
defer session.Close(ctx)

summary, err := session.ExecuteRead(ctx, func(tx neo4j.ManagedTransaction)
    result, err := tx.Run(ctx, "RETURN $answer AS answer", map[string]any{"
    if err != nil {
        return nil, err
    }

    return result.Consume(ctx)
})

if err != nil {
    log.Fatal(err)
}
```

Result Summary

The `ResultSummary` object returned by the `ExecuteRead()` and `ExecuteWrite()` methods holds information about the Cypher statement execution, including database information, execution time and in the case of a write query, statistics on changes made to the database as a result of the statement execution.

go: Result Summary

```
summaryObj := summary.(neo4j.ResultSummary)
fmt.Printf("Results available after %d ms and consumed after %d ms\n",
    summaryObj.ResultAvailableAfter(),
    summaryObj.ResultConsumedAfter())
```

Error handling

Introduction

In production applications, proper error handling is crucial for maintaining system stability and providing meaningful feedback to users.

The Neo4j Go driver provides comprehensive error handling capabilities that help you manage different types of errors gracefully.

Types of errors

Neo4j errors can be categorized into several types:

- **Connection errors** - Network issues, authentication failures
- **Constraint errors** - Violations of database constraints
- **Transaction errors** - Deadlocks, timeout issues
- **Query errors** - Syntax errors, parameter issues

go:

```go
import (
    "errors"
    "github.com/neo4j/neo4j-go-driver/v5/neo4j"
)

result, err := neo4j.ExecuteQuery(ctx, driver, query, params, neo4j.EagerRe
if err != nil {
    // Check for specific error types
    if neo4j.IsNeo4jError(err) {
        neo4jErr := err.(*neo4j.Neo4jError)
        switch neo4jErr.Code {
        case "Neo.ClientError.Schema.ConstraintValidationFailed":
            // Handle constraint violation
            log.Printf("Constraint violation: %s", neo4jErr.Msg)
        case "Neo.ClientError.Statement.SyntaxError":
            // Handle syntax error
            log.Printf("Syntax error: %s", neo4jErr.Msg)
```

Retry logic

For transient errors, you can implement retry logic:

go:

```go
func executeWithRetry(ctx context.Context, driver neo4j.Driver, query strin
    var lastErr error

    for i := 0; i < maxRetries; i++ {
        result, err := neo4j.ExecuteQuery(ctx, driver, query, params, neo4j
        if err == nil {
            return result, nil
        }

        // Check if error is retryable
        if neo4j.IsRetryable(err) {
            lastErr = err
            time.Sleep(time.Duration(i+1) * time.Second) // Exponential bac
            continue
        }

        // Non-retryable error
```