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.

Installing the Driver

To install the driver, use go get:

shell:

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

Creating a Driver Instance

You start by importing the driver and creating an instance:

go:

```go
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)
    }
```

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

Best Practice

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

```
    defer driver.Close(context.Background()) // (3)
}
```

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
```

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

```
count, _ := first.Get("count") // (5)

fmt.Println(count)
```

Full driver lifecycle

Once you have finished with the driver, call `Close()` to release any resources held by the driver.

go:

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

You can use `defer` to create an all-in-one solution that will automatically close the driver when the function exits.

go:

```
func main() {
    driver, err := neo4j.NewDriverWithContext(NEO4J_URI, neo4j.BasicAuth(NEO4J_USERNAME, NEO4J_PASSWORD, ""))
    if err != nil {
        panic(err)
    }
    defer driver.Close(context.Background())


    result, err := neo4j.ExecuteQuery(context.Background(), driver, "RETURN COUNT {()} AS count", nil, neo4j.EagerResultTransformer)
```

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 result object containing:

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

go:

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

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 m.title AS title, r.role AS role


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

Transforming results

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

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

go:

```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
    },
```

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

Working with DataFrames

The Go driver doesn't have built-in DataFrame support like Python, but you can easily transform results into structured data:

go:

```go
type MovieRole struct {
    Title string `json:"title"`
    Role  string `json:"role"`
}

result, err := neo4j.ExecuteQuery(ctx, driver,
    cypher,
    map[string]any{"name": name},
    func(result neo4j.ResultWithContext) (any, error) {
        var movies []MovieRole
        for _, record := range result.Records {
            title, _ := record.Get("title")
            role, _ := record.Get("role")
            movies = append(movies, MovieRole{
                Title: title.(string),
```

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.ExecuteQueryWithRoutingControl(neo4j.ReadRouting), // (1)
)
```

You can also use `neo4j.WriteRouting` for 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

> **Types in Neo4j Browser**
>
> When graph types are returned by a query, 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)

// (4)
name, exists := movieNode.Get("name")
if exists {
```

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 `Get()` method which returns the value and a boolean indicating if the property exists.

```
    fmt.Println(name)
}
```

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)

// 4
role, exists := relationship.Get("role")
if exists {
    fmt.Println(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. Access properties using the `Get()` method
5. `StartNodeId` - Element ID of the node at the start of the relationship
6. `EndNodeId` - Element ID of the node at the end of the relationship

```
fmt.Println(relationship.StartNodeId)       // (5)

fmt.Println(relationship.EndNodeId)         // (6)
```

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:

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

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.FixedZone("CE
        "dtstring": "2024-05-15T14:30:00+02:00",
    },
```

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 **ISO 8601 format string** within a Cypher statement
3. Get the current date and time using the `datetime()` function.

```
    neo4j.EagerResultTransformer,
)
```

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:

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

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{
```

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.

```
        "startsAt": startsAt,
        "endsAt": endsAt,
        "eventLength": eventLength,
    },
    neo4j.EagerResultTransformer,
)
```

Spatial types

Introduction

Neo4j supports spatial data types for storing and querying geographic information.

The Go driver provides support for 2D and 3D point types that can be used to represent locations on Earth or in 3D space.

Point types

Neo4j supports two types of points:

- **2D Points** - Represent locations on a 2D plane (e.g., latitude/longitude)
- **3D Points** - Represent locations in 3D space (e.g., latitude/longitude/altitude)

go:

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


// Create a 2D point (latitude, longitude)
point2D := neo4j.Point2D{
    X: -122.4194,  // longitude
    Y: 37.7749,    // latitude
    SpatialRefId: 4326, // WGS 84 coordinate system
}


// Create a 3D point (latitude, longitude, altitude)
point3D := neo4j.Point3D{
    X: -122.4194,  // longitude
```

```
    Y: 37.7749,    // latitude
    Z: 100.0       // altitude in meters
    SpatialRefId: 4979, // WGS 84 3D coordinate system
}
```

Working with spatial data

You can store and query spatial data in Neo4j:

go:

```go
// Store a location
neo4j.ExecuteQuery(ctx, driver, `
CREATE (l:Location {
    name: $name,
    coordinates: $point
})
`, map[string]any{
    "name": "San Francisco",
    "point": point2D,
}, neo4j.EagerResultTransformer)

// Query locations within a distance
result, err := neo4j.ExecuteQuery(ctx, driver, `
MATCH (l:Location)
WHERE distance(l.coordinates, point({latitude: $lat, longitude: $lon})) < $radius
```

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

A unit of work is a pattern that groups related operations into a single transaction.

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)


    if err != nil {
        return nil, err
    }


    record, err := result.Single(ctx)
    if err != nil {
        return nil, err
    }
```

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.

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

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 string, amount float64) error {
    // 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},
```

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 will be iterated over 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)
}

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.EagerResultTransformer)
if err != nil {
    // Check for specific error types
    if neo4j.IsNeo4jError(err) {
        neo4jErr := err.(*neo4j.Neo4jError)
```

Retry logic

For transient errors, you can implement retry logic:

go:

```go
func executeWithRetry(ctx context.Context, driver neo4j.Driver, query string, params map[string]any, maxRetries int) (neo4j.EagerResult, error) {
    var lastErr error
    for i := 0; i < maxRetries; i++ {
        result, err := neo4j.ExecuteQuery(ctx, driver, query, params, neo4j.EagerResultTransformer)
        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 backoff
            continue
        }
```

```
        // Non-retryable error
        return neo4j.EagerResult{}, err
    }

    return neo4j.EagerResult{}, lastErr
}
```